

09/781,158

# 日本国特許庁

PATENT OFFICE  
JAPANESE GOVERNMENT

別紙添付の書類に記載されている事項は下記の出願書類に記載されて  
いる事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed  
with this Office.

出願年月日

Date of Application: 2000年 5月 8日

出願番号

Application Number: 特願2000-135010

願

licant (s):

株式会社東芝

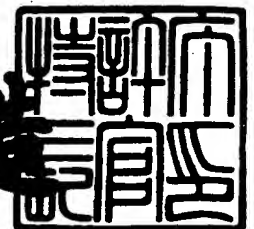
CERTIFIED COPY OF  
PRIORITY DOCUMENT

RECEIVED  
JUN 22 2001  
Technology Center 2100

2001年 2月23日

特許庁長官  
Commissioner,  
Patent Office

及川耕造



【書類名】 特許願

【整理番号】 13B004035

【提出日】 平成12年 5月 8日

【あて先】 特許庁長官殿

【国際特許分類】 H04L 9/00  
G06F 9/00

【発明の名称】 マイクロプロセッサ、これを用いたマルチタスク実行方法、およびマルチレッド実行方法

【請求項の数】 15

【発明者】

【住所又は居所】 神奈川県川崎市幸区小向東芝町 1 番地 株式会社東芝  
研究開発センター内

【氏名】 橋本 幹生

【発明者】

【住所又は居所】 神奈川県川崎市幸区小向東芝町 1 番地 株式会社東芝  
研究開発センター内

【氏名】 藤本 謙作

【発明者】

【住所又は居所】 神奈川県川崎市幸区小向東芝町 1 番地 株式会社東芝  
研究開発センター内

【氏名】 白川 健治

【発明者】

【住所又は居所】 神奈川県川崎市幸区小向東芝町 1 番地 株式会社東芝  
研究開発センター内

【氏名】 寺本 圭一

【特許出願人】

【識別番号】 000003078

【氏名又は名称】 株式会社 東芝

【代理人】

【識別番号】 100083806

【弁理士】

【氏名又は名称】 三好 秀和

【電話番号】 03-3504-3075

【選任した代理人】

【識別番号】 100068342

【弁理士】

【氏名又は名称】 三好 保男

【選任した代理人】

【識別番号】 100100712

【弁理士】

【氏名又は名称】 岩▲崎▼ 幸邦

【選任した代理人】

【識別番号】 100100929

【弁理士】

【氏名又は名称】 川又 澄雄

【選任した代理人】

【識別番号】 100108707

【弁理士】

【氏名又は名称】 中村 友之

【選任した代理人】

【識別番号】 100095500

【弁理士】

【氏名又は名称】 伊藤 正和

【選任した代理人】

【識別番号】 100101247

【弁理士】

【氏名又は名称】 高橋 俊一

【選任した代理人】

【識別番号】 100098327

【弁理士】

【氏名又は名称】 高松 俊雄

【手数料の表示】

【予納台帳番号】 001982

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 マイクロプロセッサ、これを用いたマルチタスク実行方法、およびマルチレッド実行方法

【特許請求の範囲】

【請求項 1】 プログラムごとに異なる暗号化鍵により暗号化された複数のプログラムをマイクロプロセッサ外部の記憶手段から読み出す読み出し手段と、

前記読み出し手段にて読み出した前記複数のプログラムを、それぞれ対応する復号化鍵で復号化する復号化手段と、

前記復号化された複数のプログラムを実行する実行手段と、

前記複数のプログラムのうち、一のプログラムの実行を中断する場合に、前記一のプログラムの実行状態を示す情報と、前記一のプログラムの暗号化鍵とを、前記マイクロプロセッサに固有の暗号化鍵で暗号化し、この暗号化した情報をコンテキスト情報として前記マイクロプロセッサ外部の記憶手段に書き込む実行状態書き込み手段と、

前記一のプログラムを再開する場合に、前記マイクロプロセッサに固有の暗号化鍵に対応する固有の復号化鍵で前記コンテキスト情報を復号化し、復号化された再開予定のプログラムの暗号化鍵が、前記一のプログラム本来の暗号化鍵と一致した場合にのみ、前記一のプログラムの実行を再開する再開手段と、

を具備することを特徴とする 1 チップまたは 1 パッケージのマイクロプロセッサ。

【請求項 2】 外部へ読み出すことのできない前記マイクロプロセッサ内部の記憶手段と、

前記複数のプログラムの処理対象であるデータのための暗号化属性を、前記マイクロプロセッサ内部の記憶手段に書き込む暗号化属性書き込み手段と、

前記複数のプログラムの処理対象であるデータを、前記暗号化属性に基づいて暗号化するデータ暗号化手段と、

をさらに具備し、前記暗号化属性の少なくとも一部は、前記コンテキスト情報に含まれることを特徴とする請求項 1 の 1 チップまたは 1 パッケージのマイクロプロセッサ。

【請求項 3】 前記実行状態書き込み手段は、マイクロプロセッサに固有の秘密情報に基づく署名をコンテキスト情報に付与し、

前記再開手段は、復号化されたコンテキスト情報に含まれる前記署名が、前記マイクロプロセッサに固有の秘密情報に基づく署名と一致する場合にのみ、前記一のプログラムを再開することを特徴とする請求項 1 に記載の 1 チップまたは 1 パッケージのマイクロプロセッサ。

【請求項 4】 外部へ読み出すことのできない固有の秘密鍵を内部に保持した 1 チップまたは 1 パッケージのマイクロプロセッサであって、

あらかじめ前記秘密鍵に対応する公開鍵によって暗号化されたコード暗号化鍵をマイクロプロセッサ外部の記憶手段から読み出す第 1 の読み出し手段と、

前記第 1 の読み出し手段で読み出した前記コード暗号化鍵を、前記秘密鍵を用いて復号化する第 1 の復号化手段と、

プログラムごとに異なる前記コード暗号化鍵によって暗号化された複数のプログラムを、前記マイクロプロセッサ外部の記憶手段から読み出す第 2 の読み出し手段と、

前記第 2 の読み出し手段にて読み出した前記複数のプログラムを復号化する第 2 の復号化手段と、

前記復号化された複数のプログラムを実行する実行手段と、

前記複数のプログラムのうち、一のプログラムの実行を中断する場合に、前記一のプログラムの実行状態を示す情報と、前記プログラムのコード暗号化鍵とを前記公開鍵によって暗号化し、暗号化した情報をコンテキスト情報として、前記マイクロプロセッサ外部の記憶手段に書き込む実行状態書き込み手段と、

前記一のプログラムの実行を再開する場合に、前記マイクロプロセッサ外部の記憶手段から前記コンテキスト情報を読み出し、前記秘密鍵でコンテキスト情報を復号化し、復号化されたコンテキスト情報に含まれるコード暗号化鍵が、前記一のプログラムの本来のコード暗号化鍵と一致する場合にのみ、前記一のプログラムの実行を再開する再開手段と、

を具備することを特徴とするマイクロプロセッサ。

【請求項 5】 外部へ読み出すことのできない固有の秘密鍵を内部に保持す

る 1 チップまたは 1 パッケージのマイクロプロセッサであって、

あらかじめ前記秘密鍵に対応した公開鍵によって暗号化されたコード暗号化鍵をマイクロプロセッサ外部の記憶手段から読み出す第 1 の読み出し手段と、

前記第 1 の読み出し手段で読み出した前記コード暗号化鍵を、前記秘密鍵を用いて復号化する第 1 の復号化手段と、

それぞれが異なる前記コード暗号化鍵で暗号化された複数のプログラムを、前記マイクロプロセッサ外部の記憶手段から読み出す第 2 の読み出し手段と、

前記第 2 の読み出し手段によって読み出された複数のプログラムを復号化する第 2 の復号化手段と、

前記復号化された複数のプログラムを実行する実行手段と、

前記複数のプログラムのうちのプログラムの実行を中断する場合に、一時鍵として乱数を発生し、前記一のプログラムの実行状態を示す情報を前記一時鍵で暗号化した第 1 の値と、前記一時鍵を前記一のプログラムのコード暗号化鍵で暗号化した第 2 の値と、前記一時鍵を前記マイクロプロセッサの秘密鍵で暗号化した第 3 の値とをコンテキスト情報として前記マイクロプロセッサ外部の記憶手段に書き込む実行状態書き込み手段と、

前記一のプログラムの実行を再開する場合に、前記マイクロプロセッサ外部の記憶手段から前記コンテキスト情報を読み出し、前記秘密鍵を用いて、前記コンテキスト情報に含まれる第 3 の値としての一時鍵を復号化し、復号化された一時鍵を用いて前記コンテキスト情報に含まれる実行状態情報を復号化するとともに、再開予定のプログラムのコード暗号化鍵を用いて、前記コンテキスト情報に含まれる第 2 の値としての一時鍵を復号化し、前記第 2 の値を復号化した一時鍵が、前記第 3 の値を秘密鍵で復号化した一時鍵と一致する場合にのみ前記一のプログラムの実行を再開する再開手段と、

を具備することを特徴とするマイクロプロセッサ。

【請求項 6】 前記実行状態書き込み手段は、前記マイクロプロセッサに固有の秘密鍵による署名を前記コンテキスト情報に付与し、

前記再開手段は、復号化された前記コンテキスト情報に含まれる前記署名が、前記マイクロプロセッサに固有の秘密鍵による本来の署名と一致する場合にのみ

、前記一のプログラムを再開することを特徴とする請求項 5 に記載の 1 チップまたは 1 パッケージのマイクロプロセッサ。

【請求項 7】 外部へ読み出すことのできない固有の秘密鍵を内部に保持した 1 チップまたは 1 パッケージのマイクロプロセッサであって、

あらかじめ前記秘密鍵に対応する公開鍵によって暗号化された暗号化鍵をマイクロプロセッサ外部の記憶手段から読み出す第 1 の読み出し手段と、

前記第 1 の読み出し手段で読み出した前記暗号化鍵を、前記秘密鍵を用いて復号化する第 1 の復号化手段と、

プログラムごとに異なる前記暗号化鍵によって暗号化された複数のプログラムを、前記マイクロプロセッサ外部の記憶手段から読み出す第 2 の読み出し手段と

前記第 2 の読み出し手段にて読み出した前記複数のプログラムを復号化する第 2 の復号化手段と、

前記復号化された複数のプログラムを実行する実行手段と、

外部へ読み出すことのできないマイクロプロセッサ内部の記憶手段と、

前記プログラムが処理するデータのための暗号化属性を、前記マイクロプロセッサ内部の記憶手段に書き込む暗号化属性書き込み手段と、

前記プログラムが処理するデータを、前記マイクロプロセッサ内部の記憶手段に書き込まれた暗号化属性に基づいて暗号化するデータ暗号化手段と、

前記複数のプログラムのうち、一のプログラムの実行を中断する場合に、前記一のプログラムの実行状態を示し、かつ前記暗号化属性の少なくとも一部を含む実行状態情報と、前記一のプログラムの暗号化鍵とを、前記公開鍵によって暗号化し、暗号化した情報をコンテキスト情報として、前記マイクロプロセッサ外部の記憶手段に書き込む実行状態書き込み手段と、

前記一のプログラムの実行を再開する場合に、前記秘密鍵でコンテキスト情報を復号化し、復号化されたコンテキスト情報に含まれる暗号化鍵が、前記一のプログラムの本来の暗号化鍵と一致する場合にのみ、前記一のプログラムの実行を再開する再開手段と、

を具備することを特徴とするマイクロプロセッサ。



【請求項8】 外部へ読み出すことのできない固有の秘密鍵を内部に保持した1チップまたは1パッケージのマイクロプロセッサであって、

あらかじめ前記秘密鍵に対応する公開鍵によって暗号化された暗号化鍵をマイクロプロセッサ外部の記憶手段から読み出す第1の読み出し手段と、

前記第1の読み出し手段で読み出した前記暗号化鍵を、前記秘密鍵を用いて復号化する第1の復号化手段と、

プログラムごとに異なる前記暗号化鍵によって暗号化された複数のプログラムを、前記マイクロプロセッサ外部の記憶手段から読み出す第2の読み出し手段と、

前記第2の読み出し手段にて読み出した前記複数のプログラムを復号化する第2の復号化手段と、

前記復号化された複数のプログラムを実行する実行手段と、

前記プログラムから参照されるデータのための暗号化属性と暗号化属性特定情報とが格納された、外部へ読み出すことのできないマイクロプロセッサ内部の記憶手段と、

前記暗号化属性特定情報に関連し、前記マイクロプロセッサに固有の署名を含む関連情報を、前記マイクロプロセッサ外部の記憶手段に書き込む関連情報書き込み手段と、

プログラムが参照するデータのアドレスに基づいて、前記マイクロプロセッサ外部の記憶手段から、前記関連情報を読み込む関連情報読み込み手段と、

前記読み込み手段により読み込まれた関連情報に含まれる署名を、前記秘密鍵によって検証し、マイクロプロセッサ固有の署名と一致した場合にのみ、前記暗号化属性特定情報および前記読み込まれた関連情報に基づいて、データ参照のための暗号化鍵およびアルゴリズムを決定し、前記プログラムによるデータの参照を許可する許可手段と、

前記プログラムから参照されるデータを、前記マイクロプロセッサ内部の記憶手段に記憶された前記暗号化属性に基づいて暗号化するデータ暗号化手段と、

前記複数のプログラムのうち、一のプログラムの実行を中断する場合に、前記一のプログラムの実行状態を示し、かつ前記暗号化属性の少なくとも一部が含ま

れた実行状態情報と、前記一のプログラムの暗号化鍵とを、前記公開鍵によって暗号化し、暗号化した情報をコンテキスト情報として、前記マイクロプロセッサ外部の記憶手段に書き込む実行状態書き込み手段と、

前記一のプログラムの実行を再開する場合に、前記秘密鍵でコンテキスト情報を復号化し、復号化された再開予定のプログラムの暗号化鍵が、前記一のプログラムの本来の暗号化鍵と一致する場合にのみ、前記一のプログラムの実行を再開する再開手段と、

を具備することを特徴とするマイクロプロセッサ。

【請求項 9】 前記関連情報は、前記暗号化属性特定情報と、マイクロプロセッサ固有の秘密情報と、前記関連情報のうちプロセッサ固有の署名を除く部分とに基づいて生成される第 2 の署名を含むことを特徴とする請求項 8 に記載のマイクロプロセッサ。

【請求項 10】 前記実行手段で実行されるプログラムが変更され、前記プログラムの暗号化鍵が変更された場合は、前記マイクロプロセッサ内部の記憶手段に記録された、前記プログラムから参照されるデータの暗号化属性を初期化することを特徴とする請求項 7 に記載のマイクロプロセッサ。

【請求項 11】 プログラムごとに異なる暗号化鍵により暗号化された複数のプログラムを、マイクロプロセッサ外部の記憶手段から読み出す読み出し手段と、

前記読み出し手段により読み出された前記複数のプログラムを復号化する復号化手段と、

前記復号化された複数のプログラムを実行する手段と、

第 1 の命令の実行に基づく、前記複数のプログラムの中の一のプログラムの実行中に、この実行状態を示す情報と、前記一のプログラムの暗号化鍵とを、コンテキスト情報として前記マイクロプロセッサ外部の記憶手段に書き込む書き込み手段と、

第 2 の命令の実行に基づいて、前記外部の記憶手段に書き込まれたコンテキスト情報を読み出す読み出し手段と、

前記読み出されたコンテキスト情報に含まれる前記一のプログラムの暗号化鍵

と、前記第 2 の命令の実行に基づき実行しているプログラムの暗号化鍵とを比較し、一致した場合にのみ、前記コンテキスト情報に含まれる実行状態を復元する復元手段と、

を具備することを特徴とする 1 チップまたは 1 パッケージのマイクロプロセッサ。

【請求項 1 2】 それぞれ異なる暗号化鍵により暗号化された複数のプログラムのうち、一のプログラムを、マイクロプロセッサ外部の記憶手段から読み出すステップと、

前記読み出したプログラムを、このプログラムに対応する復号化鍵で復号化し、実行するステップと、

前記一のプログラムの実行が中断された場合に、前記一のプログラムの実行状態を示す情報と、前記一のプログラムの暗号化鍵とを、前記マイクロプロセッサに固有の暗号化鍵で暗号化し、この暗号化した情報をコンテキスト情報として前記マイクロプロセッサ外部の記憶手段に書き込むステップと、

前記コンテキスト情報を、前記マイクロプロセッサに固有の暗号化鍵に対応するマイクロプロセッサに固有の復号化鍵で復号化するステップと、

復号化された再開予定のプログラムの暗号化鍵と、前記一のプログラム本来の暗号化鍵とを比較し、一致した場合にのみ、前記一のプログラムの実行を再開するステップと、

を含むマルチタスク実行方法。

【請求項 1 3】 あらかじめ異なるコード暗号化鍵で暗号化された複数のプログラムと、前記コード暗号化鍵の各々をあらかじめマイクロプロセッサに固有の公開鍵で暗号化したコード暗号化鍵とを、前記マイクロプロセッサ外部の記憶手段に格納するステップと、

前記公開鍵で暗号化されたコード暗号化鍵を、前記マイクロプロセッサ外部の記憶手段から読み出し、前記公開鍵に対応するマイクロプロセッサに固有の秘密鍵で復号化するステップと、

前記複数のプログラムを、前記マイクロプロセッサ外部の記憶手段から読み出して、前記復号化されたコード暗号化鍵で復号化するステップと、

前記復号化された複数のプログラムを実行するステップと、

前記複数のプログラムのうち、一のプログラムの実行が中断される場合に、当該一のプログラムの実行状態を示す情報と、このプログラムのコード暗号化鍵とを、前記公開鍵によって暗号化し、これをコンテキスト情報として前記マイクロプロセッサ外部の記憶手段に書き込むステップと、

前記マイクロプロセッサ外部の記憶手段から前記コンテキスト情報を読み出し、前記秘密鍵でコンテキスト情報を復号化するステップと、

前記復号化されたコンテキスト情報に含まれるコード暗号化鍵と、前記一のプログラムの本来のコード暗号化鍵とを比較し、一致する場合にのみ、前記一のプログラムの実行を再開するステップと、

を含むマルチタスク実行方法。

【請求項 1 4】 あらかじめ異なる暗号化鍵で暗号化された複数のプログラムと、前記暗号化鍵の各々をあらかじめマイクロプロセッサに固有の公開鍵で暗号化した暗号化鍵とを、前記マイクロプロセッサ外部の記憶手段に格納するステップと、

前記複数のプログラムの各々の処理対象となるデータのための暗号化属性を、外部へ読み出すことのできないマイクロプロセッサ内部の記憶手段に書き込むステップと、

前記公開鍵で暗号化された暗号化鍵を、前記マイクロプロセッサ外部の記憶手段から読み出し、前記公開鍵に対応するマイクロプロセッサに固有の秘密鍵で復号化するステップと、

前記暗号化された複数のプログラムを、前記マイクロプロセッサ外部の記憶手段から読み出し、前記復号化された暗号化鍵で復号化するステップと、

前記復号化された複数のプログラムを実行する実行するステップと、

前記複数のプログラムのうち、一のプログラムの実行を中断する場合に、前記一のプログラムの実行状態を示し、かつ前記暗号化属性の少なくとも一部を含む実行状態情報と、前記一のプログラムのコード暗号化鍵とを、前記公開鍵によって暗号化し、これをコンテキスト情報として、前記マイクロプロセッサ外部の記憶手段に書き込むステップと、

前記プログラムの処理対象であるデータを、前記マイクロプロセッサ内部の記憶手段に書き込まれた暗号化属性に基づいて暗号化し、暗号化されたデータを前記マイクロプロセッサ外部の記憶手段に保存するステップと、

前記コンテキスト情報を前記マイクロプロセッサの秘密鍵で復号化するステップと、

前記復号化されたコンテキスト情報に含まれる暗号化鍵が、前記一のプログラムの本来の暗号化鍵とを比較し、一致する場合にのみ、前記一のプログラムの実行を再開するステップと、

を含むマルチタスク実行方法。

【請求項 1 5】 それぞれ異なる暗号化鍵であらかじめ暗号化された複数のプログラムを、マイクロプロセッサ外部の記憶手段に格納するステップと、

前記複数のプログラムを前記マイクロプロセッサ外部の記憶手段から読み出して、復号化してから実行するステップと、

第 1 の命令の実行に基づき、前記複数のプログラムの中の一のプログラムの実行中に、この実行状態を示す情報と、前記一のプログラムの暗号化鍵とを、コンテキスト情報として前記マイクロプロセッサ外部の記憶手段に書き込むステップと、

第 2 の命令の実行に基づいて、前記マイクロプロセッサ外部の記憶手段に書き込まれたコンテキスト情報を読み出すステップと、

前記読み出されたコンテキスト情報に含まれる前記一のプログラムの暗号化鍵と、前記所定の第 2 の命令を実行中のプログラムの暗号化鍵とを比較し、一致した場合にのみ、前記コンテキスト情報に含まれる実行状態を復元するステップと

を含むマルチスレッド実行方法。

【発明の詳細な説明】

【 0 0 0 1 】

【発明の属する技術分野】

本発明は、マルチタスクのプログラム実行環境下で、実行コードや処理対象であるデータの不正な改変を防止することのできるマイクロプロセッサ、これを用

いたマルチタスク実行方法、およびマルチレッド実行方法に関する。

【0002】

【従来の技術】

近年マイクロプロセッサの性能向上は著しく、従来の用途である計算やグラフィックに留まらず動画像、音声の再生や編集加工が可能となっている。これによりマイクロプロセッサを応用したエンドユーザ向けシステム（以下PCと呼ぶ）でユーザはさまざまな動画像や音声を楽しめるようになった。PCを動画像、音声の再生装置として使うことは、PCが持っている計算能力と組み合わせてゲームなどの応用ができることは当然として、固定的なハードウェアを必要としないため、すでにPCを持っているユーザにとっては安価に動画像、音声の再生が楽しめるという利点もある。

【0003】

しかしながらPCで画像／音声を扱う場合に問題となるのがその画像／音声の著作権の保護である。MDやデジタルビデオデッキでは、不正なコピーを防ぐための機構がそれら装置に組み込まれることによって無制限なコピーが防止されていた。そこでは、装置を改造して不正コピーを行うことは極めて稀であり、また仮に改造された装置があったとしてもそれが広く流通することは充分防止できるという仮定がなされていた。実際、世界的に見て不正コピーを目的とする装置の製造、販売は法律によって禁止される方向にあり、不正コピーによる被害は著作権者の収益を大きく脅かすほどの問題とはなっていない。

【0004】

ところが、PC上で画像、音楽の再生を行うときにデータを扱うのではソフトウェアである。PCではエンドユーザがソフトウェアの改変を自由に行うことができってしまう。ソフトウェアの場合、ある程度の知識があればプログラムを解析することにより、それらソフトウェアを改造して不正コピーをすることが可能であることは充分想像できる。さらに問題なのはそうして作られて不正コピーのソフトウェアはハードウェアと比べて、ネットワークなどの媒体を通じて広まりやすいことがある。

【0005】

この問題を解決するために、著作権が問題となる商業映画や音楽の再生に使われるPCソフトウェアは、ソフトウェアを暗号化するなどの手法により、解読、改竄されることを防止する技術が使われている。この技術は耐タンパソフトウェア技術と呼ばれている (David Aucsmith et.al; "Tamper Resistant Software: An Implementation", Proceeding of the 1996 Intel Software Developer's Conference)。

【 0 0 0 6 】

また、動画像、音声に限らずPCを通じてユーザに提供される著作物あるいはノウハウとして価値のある情報を不正にコピーされたり、PCソフトウェア自体に含まれるノウハウなどを解析から守るのにも耐タンパソフトウェア技術は有効である。

【 0 0 0 7 】

しかし、耐タンパソフトウェア技術は、プログラムのうち保護を要する部分を実行開始時には暗号化しておき、その部分を実行する前に復号化し、実行終了後に再び暗号化することにより、逆アセンブラ、デバッガなどの解析ツールによる解析を困難にするという技術である。したがって、プログラムがプロセッサによって実行可能である以上、プログラムの開始時から順を追って解析していけば必ず解析することが可能になる。

【 0 0 0 8 】

これは、著作権者がPCを利用した動画像、音声を再生するシステムに著作物を提供する際の妨げとなっている。

【 0 0 0 9 】

また、その他の耐タンパソフトの応用についても同様の弱点があり、PCを通じた高度な情報サービスや企業、個人のノウハウを含んだプログラムのPCへの適用の妨げにもなっている。

【 0 0 1 0 】

上記のことはソフトウェアの保護一般にあてはまる問題であるが、これに加えて、オープンプラットフォームであるPCには、システムのソフトウェア的土台となるはずのオペレーションシステム (以下、適宜「OS」と略する) 自体が改

変されて攻撃の手段とされるという問題もある。技術を持ち、悪意のあるユーザは、自分が所有するPCのOSに改造を加え、OSの持つ特権を利用して、OSにアプリケーションプログラムに埋め込まれている著作権保護機構を無効化させたり、その機構を解析させたりできるからである。

## 【 0 0 1 1 】

現代のOSは、CPUに備えられた実行制御機能やメモリへの特権操作機能を利用して、コンピュータの制御下にある資源の管理や利用の調停を行なっている。管理の対象として、従来からの対象であるデバイスや、CPU、メモリ資源に加え、ネットワークやアプリケーションレベルのQoS (Quality of Service) などが加わってきている。とはいえ、資源管理の基本はあくまでも、プログラムの実行に必要な資源の割り当てである。すなわち、そのプログラムの実行に対するCPU時間の割り当てと、実行に必要なメモリ空間の割り当てが、資源管理の基本となる。それ以外のデバイス、ネットワーク、アプリケーションQoSの制御は、これらの資源に対してアクセスするプログラムの実行を制御することによって（CPU時間とメモリを割り当てることによって）行われる。

## 【 0 0 1 2 】

OSは、CPU時間の割り当てと、メモリの割り当てを行うための特権を有する。すなわち、CPU時間の割り当てをするために、アプリケーションプログラムを任意の時点で停止、再開する特権と、任意の時点でアプリケーションに割り当てたメモリ空間の内容を、異なる階層のメモリに移動する特権である。後者の特権は、異なるアクセス速度と容量を持つ（通常は）階層化されたメモリシステムをアプリケーションから隠ぺいして、フラットなメモリ空間をアプリケーションに提供するためにも使用される。

## 【 0 0 1 3 】

この2つの特権を持つことにより、任意の時点でOSはアプリケーションの実行状態を停止してスナップショットをとり、それをコピーまたは書き換えて再開することが可能である。この機能はアプリケーションのもつ秘密を解析する道具にも使うことができる。

## 【 0 0 1 4 】



コンピュータ上のアプリケーションに対する解析を防ぐために、プログラムやデータの暗号化を行う技術はすでにいくつか知られている（Hampsonによる米国特許第4,47,902号、Hartmanによる米国特許第5,224,166号、Davisによる米国特許第5,806,706号、Takahasiによる米国特許第5,825,878号、Leonard et.alによる米国特許第6,003,117号、特開平11-282756号公報等）。しかし、これらの技術では、上述したようなオペレーションシステムの特権動作からプログラムの動作とデータの秘密を守ることは考慮されていない。

## 【0015】

Intel社のx86アーキテクチャに基づく従来技術（Hartmanの米国特許第5,224,166号、“System for seamless processing of encrypted and non-encrypted data and instructions”、以下、「従来技術1」とする）は、実行コードとデータを、予め定められた暗号化鍵 $K_x$ で暗号化してメモリに格納する技術を開示している。 $K_x$ は、プロセッサに埋め込まれた秘密鍵 $K_s$ に対応する公開鍵 $K_p$ で暗号化されて $E_{K_p}[K_x]$ の形で与えられる。したがって、 $K_s$ を知っているプロセッサのみがメモリ上の暗号化された実行コードを復号化できる。暗号化鍵 $K_x$ は、セグメントレジスタと呼ばれるプロセッサ内部のレジスタに格納される。

## 【0016】

この機構により、確かにプログラムコードを暗号化することでユーザからコードの秘密をある程度保護することは可能である。また、コードの暗号化鍵 $K_x$ を知らない者がコードを意図した通りに改変することや、暗号化鍵 $K_x$ で復号化されて実行可能なコードを新たに作成することは暗号学的に困難となる。

## 【0017】

## 【発明が解決しようとする課題】

しかし、この技術を適用したシステムでは、実行コードの暗号化を解かなくても、コンテキスト切替と呼ばれるOSが持つ特権を利用することで、プログラムの解析が可能になるという欠点がある。

## 【0018】

具体的には、割り込みによってプログラムの実行が中断された時や、プログラ

ムがシステムコール呼び出しのために自発的にソフトウェア割り込み命令を呼び出した時、OSは他のプログラム実行のために、コンテキストの切替処理を行う。コンテキストの切替とは、その時点でのレジスタ値の集合からなるプログラムの実行状態（以下コンテキスト情報と呼ぶ）をメモリに保存し、あらかじめメモリに保存されていた別のプログラムのコンテキスト情報をレジスタに復帰させる操作である。x86プロセッサにおける従来のコンテキスト保存の形式を、図18に示す。ここにはアプリケーションが使うレジスタの内容が全て含まれている。中断されたプログラムのコンテキスト情報は、プログラムが再開される時に元のレジスタに復帰される。コンテキスト切替は複数のプログラムを並行動作させる上で不可欠の機能である。従来の技術では、OSはコンテキスト切替の時にレジスタ値を読むことができるので、そのプログラムの実行状態がどのように変化したかに基づいて、全てとは限らないが、プログラムの行なっている動作のほとんどの部分を推定することができる。

## 【0019】

さらにタイマなどの設定によって例外を発生させるタイミングを制御すれば、プログラムの任意の実行時点でこの処理を行うことができる。また、実行を中断、解析するばかりでなく、レジスタの情報を悪意で書き換えることも可能である。レジスタの書き換えは、プログラムの動作を変更できるばかりでなく、プログラムの解析も容易にする。OSはアプリケーションの任意の状態を保存することができるため、ある時点で保存した状態から、繰り返し何度もレジスタ値を書き換えてプログラムを動作させてプログラムの動作を解析することが可能なのである。上記の機能に加えてプロセッサにはステップ実行などのデバッグ支援機構があり、OSはこれらの機能を全て利用してアプリケーションの解析を行うことができるという失点がある。

## 【0020】

また、前記従来技術1ではデータに関しては、暗号化されたコードセグメントを通じたプログラム実行によってのみ、プログラムは暗号化データにアクセスできるとされている。このとき、それぞれ異なる暗号化鍵で暗号化されたプログラムがあっても、暗号化されたデータは、プログラムがどのような暗号化鍵で暗号

化されているかにかかわらず、任意の鍵を用いて、暗号化されたプログラムから自由に読むことができるという問題がある。ここに示した従来技術では、OSとアプリケーションが独立に秘密を持ち、アプリケーションの秘密をOSから守ることや、複数のプログラム供給者がそれぞれに秘密を持つことは考えられていないのである。

## 【 0 0 2 1 】

もちろん、仮想記憶機構に設けられた保護機能によって、アプリケーションの間でメモリ空間を隔離したり、アプリケーションによるシステムメモリへのアクセスを禁止することは既存のプロセッサでも可能である。しかし、仮想記憶機構がOSの管理下にある以上、アプリケーションの秘密を守ることをOSの管理下ある機能に依存することはできない。OSは保護機構を無視してデータをアクセスすることができるからであり、前述のように仮想記憶機能の提供においてその特権は不可欠である。

## 【 0 0 2 2 】

別の従来技術として、特開平 1 1 - 2 8 2 7 5 6（以下従来技術 2 と呼ぶ）では、アプリケーションの持つ秘密情報を保存するためにCPU内に設けられた秘密メモリの技術が開示されている。この例では秘密メモリの中のデータへアクセスするために予め定められた参照値を必要とするとしている。しかしながら、同一のCPUで動作する複数のプログラム、特にOSから、秘密データのアクセス権を得るための参照値をどのように保護するかはこの公知技術においては開示されていない。

## 【 0 0 2 3 】

さらに、米国特許第5,123,045号（Ostorovsky et.al、以下従来技術 3 と呼ぶ）では、アプリケーション対応に固有の秘密鍵を持つサブプロセッサを前提とし、それらサブプロセッサがメインメモリ上に置かれたプログラムをアクセスする時にそのアクセスパターンからプログラムの動作を推定されることのないシステムが開示されている。メモリに対して操作を行う命令体系を、それとは異なる命令体系に変換し、ランダムなメモリアクセスが行われるのがその機構である。

## 【 0 0 2 4 】

しかしながら、この技術はアプリケーション毎に異なるサブプロセッサが必要となり、コスト高になるばかりでなく、このような命令体系を処理するコンパイラやプロセッサハードウェアの実装、高速化技術は、現在のプロセッサのそれとはかなり異なり、非常に困難なものとなることが予想される。なによりも、このようなプロセッサでは、実際に動作するコードの動作やデータを観察、追跡してもデータの内容や動作の対応関係の把握が著しく困難となるため、プログラムコードやデータが単純に暗号化されている前の公知技術 1, 2 と比較して、プログラムのデバッグが非常に困難となる点で、実用には多くの課題がある。

## 【 0 0 2 5 】

そこで、これらの問題点を解決するために、本発明の第1の目的は、マルチタスク環境下において、割り込みにより実行が中断された場合でも、内部の実行アルゴリズムとメモリ領域内部のデータ状態の双方を、不正な解析から確実に保護することのできるマイクロプロセッサの提供にある。この目的は、従来技術ではプログラムコードの値を保護することはできたが、例外発生やデバッグ機能によるプログラム実行の中断を利用した解析を防ぐことはできなかったという問題に鑑みている。そこで、プログラム実行中断時にも、確実にコードを保護することができ、かつ、その保護が、現代のOSが必要とする実行制御機能およびメモリ管理機能の双方と両立するマイクロプロセッサの提供を目的とする。

## 【 0 0 2 6 】

本発明の第2の目的は、異なる暗号化鍵で暗号化された複数のプログラムが実行される場合にも、それぞれのプログラムが互いに独立して、正しく読み書きできるデータ領域を確保したマイクロプロセッサの提供にある。特に、Hartmanによる米国特許第5,224,166号（従来技術1）では、暗号化されていないコードによる暗号化されたデータ領域へのアクセスが禁止されるだけの単純な保護しか行なわれておらず、複数のプログラムがそれぞれ独立に秘密を保護することができなかった。この問題点に鑑みて、複数のアプリケーションがそれぞれ秘密を持つ（暗号化されている）場合、各々の秘密をOSから守るデータ領域を有するマイクロプロセッサの提供を第2の目的とする。

## 【 0 0 2 7 】

本発明の第3の目的は、第2の目的で確保したデータ領域の保護属性（すなわち暗号化属性）を、OSによる不正な書き換えから保護することのできるマイクロプロセッサの提供にある。従来技術1では、プログラムがメモリ上にデータを書きだす直前に、OSがコンテキスト切替を利用してプログラムの実行を中断し、セグメントレジスタに設定された暗号化属性を書き換えることができるという欠点があった。暗号化属性を書き換えることによって平文でデータが書き込まれる状態になると、データは暗号化されずにメモリに書き込まれてしまう。アプリケーションがあるタイミングでセグメントレジスタ値をチェックしたとしても、その後にレジスタ値が書き換えられてしまえば同じことである。そこで、このような改変を禁止または検出して対抗措置をとることのできる機構を備えたマイクロプロセッサの提供を第3の目的とする。

## 【0028】

本発明の第4の目的は、プログラムがデータ暗号化鍵として任意の値を使用することができ、暗号解析理論上のいわゆる「選択平文攻撃」から暗号化属性を守ることのできるマイクロプロセッサの提供にある。

## 【0029】

本発明の第5の目的は、プログラムデバッグ、フィードバックのための機構を備えたマイクロプロセッサを提供することにある。すなわち、マイクロプロセッサ自体に平文でプログラムのデバッグを行わせ、また、実行に失敗した場合にプログラムコードの提供者（プログラムベンダ）に不具合情報をフィードバックさせることを意図する。

## 【0030】

本発明の第6の目的は、上記第1～第5の目的を、安価かつ高性能に実現することのできるマイクロプロセッサの提供にある。

## 【0031】

## 【課題を解決するための手段】

本発明の第1の特徴として、上記第1の目的を達成するために、1チップまたは1パッケージとして構成されるマイクロプロセッサは、読み出し手段としてのバスインターフェイスユニットを介して、プログラムごとに異なるコード暗号化

鍵で暗号化された複数のプログラムを、マイクロプロセッサ外部の記憶手段（たとえばメインメモリ）から読み出す。マイクロプロセッサの復号化手段は、読み出した複数のプログラムを、それぞれ対応する復号化鍵で復号化し、命令実行部が、復号化された複数のプログラムを実行する。複数のプログラムのうち、あるプログラムの実行を中断する場合に、実行状態書き込み手段としてのコンテキスト情報暗号化／復号化ユニットは、中断されるプログラムのそれまでの実行状態を示す情報と、このプログラムのコード暗号化鍵とを、マイクロプロセッサに固有の暗号化鍵で暗号化し、暗号化した情報をコンテキスト情報としてマイクロプロセッサ外部の記憶手段に書き込む。中断されたプログラムを再開する場合に、再開手段としての検証ユニットは、暗号化されたコンテキスト情報を、マイクロプロセッサに固有の暗号化鍵に対応する固有の復号化鍵で復号化し、復号化されたコンテキスト情報に含まれるコード暗号化鍵（すなわち再開予定のプログラムのコード暗号化鍵）が、中断されたプログラム本来のコード暗号化鍵と一致した場合にのみ、プログラムの実行を再開する。

#### 【0032】

また、第2および第3の目的を達成するために、マイクロプロセッサは、外部へ読み出すことのできないプロセッサ内部の記憶領域（たとえばレジスタ）と、プログラムの処理対象であるデータのための暗号化属性を、内部の記憶手段に書き込む暗号化属性書き込み手段（たとえば命令TLB）とを、さらに有する。暗号化属性とは、たとえば、プログラムのコード暗号化鍵と、暗号化の対象となるアドレス範囲である。このような暗号化属性の少なくとも一部は、コンテキスト情報に含まれる。

#### 【0033】

コンテキスト情報暗号／復号化ユニットはまた、マイクロプロセッサに固有の秘密情報に基づく署名をコンテキスト情報に付与する。この場合、検証ユニットは、復号化されたコンテキスト情報に含まれる署名が、マイクロプロセッサに固有の秘密情報に基づく本来の署名と一致するかどうか判断し、一致する場合にのみ中断されていたプログラムを再開する。

#### 【0034】

このように、暗号化プログラムによるそれまでの実行状態は、コンテキスト情報として外部メモリに保存されるとともに、実行処理対象であるデータの保護属性は、プロセッサ内部のレジスタに格納されているので、データに対する不正な改変が防止される。

## 【 0 0 3 5 】

本発明の第2の特徴として、上記第4の目的を達成するために、1チップまたは1パッケージとして構成されるマイクロプロセッサは、外部へ読み出すことのできない固有の秘密鍵を内部に保持する。読み出し手段としてのバスインターフェイスユニットは、あらかじめ秘密鍵に対応したマイクロプロセッサに固有の公開鍵によって暗号化されたコード暗号化鍵を、マイクロプロセッサ外部の記憶手段から読み出す。第1の復号化手段としての鍵復号化ユニットは、読み出したコード暗号化鍵を、マイクロプロセッサの秘密鍵を用いて復号化する。バスインターフェイスユニットはまた、それぞれが異なるコード暗号化鍵で暗号化された複数のプログラムを、外部の記憶手段から読み出す。第2の復号化手段としてのコード復号化ユニットは、読み出された複数のプログラムを復号化する。命令実行部は、復号化された複数のプログラムを実行する。複数のプログラムのうち、あるプログラムの実行が中断される場合に、乱数発生機構は、一時的な鍵として乱数を発生する。コンテキスト情報暗号／復号化ユニットは、中断されるプログラムの実行状態を示す情報を生成された乱数で暗号化した第1の値と、この乱数を中断するプログラムのコード暗号化鍵で暗号化した第2の値と、乱数をマイクロプロセッサの秘密鍵で暗号化した第3の値とを、コンテキスト情報として外部の記憶手段に書き込む。

## 【 0 0 3 6 】

プログラムの実行を再開するときは、コンテキスト情報暗号／復号化ユニットは、外部の記憶手段から前記コンテキスト情報を読み出し、秘密鍵を用いて、コンテキスト情報に含まれる第3の値の乱数を復号化し、復号化された乱数を用いて、コンテキスト情報に含まれる実行状態情報を復号化する。同時に、再開される予定のプログラムのコード暗号化鍵を用いて、コンテキスト情報に含まれる第2の値の乱数を復号化する。第2の値をコード暗号化鍵で復号化した乱数と、第

3の値を秘密鍵で復号化した一時鍵とを比較し、一致する場合にのみ、プログラムの実行を再開する。

【0037】

このように、それまで実行されていた状態を示すコンテキスト情報は、保存の都度生成される乱数で暗号化され、マイクロプロセッサに固有の秘密鍵による署名が添付されるので、安全なかたちで外部メモリに保存される。

【0038】

本発明の第3の特徴として、前記第1～第3および第6の目的を達成するために、1チップまたは1パッケージとして構成されるマイクロプロセッサは、プログラムごとに異なる前記暗号化鍵によって暗号化された複数のプログラムを、マイクロプロセッサ外部の記憶手段から読み出して実行する。このマイクロプロセッサは、外部へ読み出すことのできない内部記憶手段（たとえばレジスタ）を有し、レジスタに、各プログラムから参照される（すなわち処理対象となる）データのための暗号化属性と暗号化属性特定情報とを格納する。コンテキスト情報暗号／復号化ユニットは、レジスタに格納された暗号化属性特定情報と関連し、マイクロプロセッサに固有の署名を含む関連情報を、外部の記憶手段に書き込む。保護テーブル管理部は、プログラムが参照するデータのアドレスに基づいて、外部の記憶手段から関連情報を読み込む。検証ユニットは、読み込まれた関連情報に含まれる署名を、秘密鍵によって検証し、それがマイクロプロセッサ固有の署名と一致した場合にのみ、暗号化属性特定情報と読み込まれた関連情報に基づいて、プログラムによるデータの参照を許可する。

【0039】

この構成では、内部レジスタに格納されるべき情報に署名をつけて、外部メモリに書き込んでおき、必要な部分だけをマイクロプロセッサに読み込む。読み込み時に、署名を検証するので、すり替えなどに対する安全性が確保される。扱うプログラムの数が増え、暗号化属性の種類が増大した場合にも、マイクロプロセッサ内部の記憶領域を拡張する必要がなく、コストを低減できる。

【0040】

本発明の、その他の特徴、効果は、図面を参照して以下で述べる詳細な説明に



より、明らかになるものである。

【0041】

【発明の実施の形態】

(第1実施形態)

図1～14を参照して、本発明の第1実施形態にかかるマイクロプロセッサについて説明する。本実施形態では、広く普及しているIntel社のPentium Proマイクロプロセッサ（インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル参照）に変更を加えたアーキテクチャを例にとって本発明の構成を説明するが、これは本発明の適用範囲を前記プロセッサに制限するものではない。なお、Intel Pentium Proプロセッサに特徴的な部分や、他のアーキテクチャへの適用については、可能な限り本文中で補足する。

【0042】

また、Pentium Proアーキテクチャではアドレス空間に物理アドレス、リニアアドレス、論理アドレスの3種類の区分があるが、本実施例ではPentiumの用語におけるリニアアドレスを、論理アドレスという呼び方で使用するものとする。

【0043】

以下の説明では、特に断らない限り、保護とはアプリケーションの秘密の保護（すなわち暗号化による保護）を意味する。したがって、通常使われる保護の概念、すなわちあるアプリケーションの動作によって他のアプリケーションの動作が妨害されないこととは区別される。ただし、本発明においては、アプリケーションの秘密の保護と並列して、通常の意味での動作保護機構がオペレーションシステム（OS）によって当然に提供されているものとする（ただし、後者に関しては本発明と直接の関係がないので、説明は省略する。）。

【0044】

また、以下の実施形態では、秘密保護をOSの管理下のアプリケーションの秘密を守るものとして説明しているが、この機構はOSそのものを改竄や解析から守る機構としても利用可能である。

【0045】

図1は本発明の第1実施形態にかかるマイクロプロセッサの基本構成図であり

、図2は、図1に示したマイクロプロセッサの詳細構成図である。

【0046】

マイクロプロセッサ101は、プロセッサコア111、命令TLB（変換索引バッファ）121、例外処理部131、データTLB141、2次キャッシュ152を含む。プロセッサコア111は、バスインタフェースユニット112、コード・データ暗号化／復号化処理部113、1次キャッシュ114、および命令実行部115を有する。命令実行部115は、命令フェッチ／デコードユニット214と、命令プール215と、命令実行切替ユニット216と、命令実行完了ユニット217を有する。

【0047】

例外処理部131は、レジスタファイル253、コンテキスト情報暗号／福号化ユニット254、例外処理ユニット255、秘密保護違反検出ユニット256、コード暗号化鍵・署名検証ユニット257を含む。命令TLB121は、ページテーブルバッファ230と、コード復号化鍵バッファ231と、鍵復号化ユニット232を有する。データTLB141は、保護テーブル管理部233を有する。

【0048】

マイクロプロセッサ101は、このマイクロプロセッサに固有の公開鍵 $K_p$ と秘密鍵 $K_s$ を保存する鍵記憶領域241を有する。今、あるプログラムベンダから所望の実行プログラムAを購入し、実行する場合を考える。プログラムベンダは、実行プログラムAを供給する前に、コード暗号化の共通鍵 $K_{code}$ でプログラムAを暗号化し( $E_{K_{code}}[A]$ )、暗号化に用いた共通鍵 $K_{code}$ をマイクロプロセッサ101の公開鍵 $K_p$ で暗号化して( $E_{K_p}[K_{code}]$ )マイクロプロセッサ101に送ってくる。マイクロプロセッサは、この実行プログラムAだけではなく、複数の異なる暗号化プログラムを擬似並列的に（すなわち割り込み中断可能に）処理するマルチタスクプロセッサである。また、暗号化プログラムだけではなく、平文プログラムの実行も当然行う。

【0049】

マイクロプロセッサ101は、バスインターフェイスユニット（読み出し手段

） 1 1 2 を介して、プログラムごとに異なるコード暗号化鍵で暗号化された複数のプログラムを、マイクロプロセッサ外部のメインメモリから読み出す。コード復号化ユニット 2 1 2 は、読み出した複数のプログラムを、それぞれ対応する復号化鍵で復号化し、命令実行部 1 1 5 が、復号化された複数のプログラムを実行する。あるプログラムの実行を中断する場合に、例外処理部 1 3 1 のコンテキスト情報暗号化／復号化ユニット 2 5 4 は、中断されるプログラムのそれまでの実行状態を示す情報と、このプログラムのコード暗号化鍵とを、マイクロプロセッサの公開鍵で暗号化し、暗号化した情報をコンテキスト情報としてメインメモリ 2 8 1 に書き込む。中断されたプログラムを再開する場合は、コード暗号化鍵・署名検証ユニット 2 5 7 は、暗号化されたコンテキスト情報を、マイクロプロセッサの秘密鍵で復号化し、復号化されたコンテキスト情報に含まれるコード暗号化鍵（すなわち再開予定のプログラムのコード暗号化鍵）が、中断されたプログラム本来のコード暗号化鍵と一致するかどうかを検証し、一致した場合にのみ、プログラムの実行を再開する。

#### 【 0 0 5 0 】

ここで、マイクロプロセッサ 1 0 1 の詳細な構成、機能を説明する前に、まずマイクロプロセッサ 1 0 1 による平文の命令の実行と、暗号化プログラムの実行における処理の流れをおおまかに説明する。

#### 【 0 0 5 1 】

マイクロプロセッサ 1 0 1 が平文命令を実行する時は、命令フェッチ／デコードユニット 2 1 4 が、プログラムカウンタ（不図示）の示すアドレスの内容を L 1 命令キャッシュ 2 1 3 から読み出そうとする。指定のアドレスの内容がキャッシュされていれば L 1 命令キャッシュ 2 1 3 から命令を読み出し、命令プール 2 1 5 に送って命令が実行される。命令プールは複数の命令を並列実行可能であり、実行を行うためのデータの読み出しを、命令実行切替ユニット 2 1 6 に要求してデータを受け取る。命令が並行して実行され、その実行結果が確定すれば、実行結果は命令実行完了ユニット 2 1 7 に送られる。命令実行完了ユニット 2 1 7 は、操作対象がマイクロプロセッサ 1 0 1 内部のレジスタであれば、実行結果をレジスタファイル 2 5 3 に書き込み、操作対象がメモリであれば、L 1 データキ

キャッシュ 218 に書き込む。

【0052】

L1 データキャッシュ 218 の内容は、バスインタフェースの制御下にある L2 キャッシュ 152 でさらにもう一度キャッシュされて、メインメモリ 281 に書き込まれる。ここでは仮想記憶機構が使われており、論理的なメモリアドレスと物理的なメモリアドレスとの対応関係を定義するのが図 3 に示すページテーブルである。ページテーブルは、物理メモリ上におかれるデータ構造である。データ TLB 141 は、実際に論理アドレスから物理アドレスへの変換を行うと同時に、データキャッシュを管理する。データ TLB 141 は、マイクロプロセッサ 101 内部のあるレジスタが示すテーブルの先頭アドレスに基づいて、テーブルの必要な部分を読み込み、論理アドレスから物理アドレスへの変換作業を行う。このとき、メモリ上のページテーブルの全てがデータ TLB 141 に読み込まれるのではなく、アクセスされる論理アドレスに応じて必要な部分だけがページテーブルバッファ 234 に読み込まれる。

【0053】

キャッシュ動作の基本は、プログラムの命令が暗号化されているかどうかにかかわらず、一定である。すなわち、命令 TLB 121 にページテーブルの一部が読み込まれ、その定義に従ってアドレス変換が行われる。バスインタフェースユニット 112 は、メインメモリ 281 または L2 キャッシュ 141 から命令を読み込み、L1 命令キャッシュ 213 に命令が格納される。L1 命令キャッシュ 213 への命令の読み出しは、複数のワードで構成されるラインと呼ばれる単位で行われ、ワード単位の読み出しよりも高速なアクセスが行われる。

【0054】

実行命令の演算処理対象であるデータについても、物理メモリ上にある同じページテーブルを利用してアドレス変換が行われるが、変換の実行は上述したようにデータ TLB 141 で行われる。

【0055】

ここまでは一般のキャッシュメモリの動作と基本的に同一であり、プロセッサの動作とキャッシュメモリについての詳細は、Curt Schimmel 著の「UNIX 力

ーネル内部解析」(ソフトバンク社、1996年)に記載されている。

【0056】

次に、暗号化されたプログラムを実行する場合の動作を説明する。本発明では秘密の保護を受ける実行コードは全て暗号化されていることを前提とし、暗号化されている実行コードのことを保護されたコードとも言う。さらに、同一の暗号化鍵による保護の範囲を保護ドメインと呼ぶ。すなわち、同じ鍵で保護されるコードの集合は同じ保護ドメインに所属し、異なる暗号化鍵で保護されるコードは保護ドメインが異なるという。

【0057】

まず、メインメモリ281上には、共通鍵方式のブロック暗号アルゴリズムによって暗号化されたプログラムの実行コードが格納されている。プログラムベンダから送られてきた暗号化プログラムのロード方法などについては、同一出願人による特許出願第2000-35898号に記載されるとおりである。暗号ブロックサイズは、ブロックサイズの2のべき乗倍が、キャッシュメモリの読み書きの単位であるラインサイズと一致すれば、どのような値をとってもよい。ただし、ブロックサイズが小さい場合、ブロック長と命令長が一致して、サブルーチンの先頭部分などの予測可能な命令部分と暗号化データの対応関係を記録することによって、簡単に命令が解読されてしまう危険がある。そこで、本発明ではブロックをインタリーブして、ブロック中のデータの間に依存性を持たせ、暗号化されたブロックに複数の命令語やオペランドの情報を含ませる。こうすることにより、命令と暗号化ブロックの対応づけを困難にする。

【0058】

図5は、本発明で行うインタリーブの例を示す。図5に示す例では、キャッシュのラインサイズを32バイト、ブロックサイズを64ビット(すなわち8バイト)としている。インタリーブ前は1ワードは4バイトで構成され、ワードAはA0~A3の4バイトからなる。1ラインはA~Hの8ワードで構成されている。これをブロックサイズ64ビットに対応する8バイト単位になるようにインタリーブすると、図5の下の方のように、A0, B0, ..., H0がワード0および1に該当する最初のブロックに配置され、A1, B1, ..., H1が次のブロック

に配置される。

#### 【 0 0 5 9 】

インタリーブを施す領域の長さは、長ければ長いほど攻撃は困難となるが、ラインサイズを越えてインタリーブをかけることは、あるキャッシュラインの復号化／暗号化が別のラインの読み出し／書き込みに依存することになり、処理の複雑化と速度低下を招くことになる。インタリーブを施す範囲はキャッシュラインサイズの範囲内にとどめることが望ましい。

#### 【 0 0 6 0 】

ここではキャッシュラインに含まれる複数のブロックのデータに依存性を持たせるためにブロックのデータをインタリーブする方法をとっているが、データブロック間に依存性を持たせるための他の方法、例えばブロック暗号のCBC (Cipher Block Chaining) モードなどを使ってもよい。

#### 【 0 0 6 1 】

暗号化された実行コードの復号化鍵K code (共通鍵アルゴリズムでは暗号化鍵と復号化鍵は同一なので、以下では復号に用いる場合も暗号化鍵という言葉を使う) は、ページテーブルに基づいて決定される。図3および4は、論理アドレスから物理アドレスへの変換テーブル構造を示す。

#### 【 0 0 6 2 】

プログラムカウンタの論理アドレス301がある値を示し、その上位ビットのディレクトリ302とテーブル303によって、ページエントリ307-jが指定される。ページエントリ307-jは、キーエントリID307-j-Kを含み、このIDに基づいて、キーテーブル309内で、このページの復号化に使用される鍵エントリ309-mが決定される。キーテーブル309の物理アドレスは、マイクロプロセッサ内部のキーテーブル制御レジスタ308によって指定される。

#### 【 0 0 6 3 】

この構成では、ページエントリに直接鍵情報をおかずに、鍵エントリのIDを置くことにより、サイズの大きい鍵情報を複数のページで共有して、サイズの限られる命令TLB121上のメモリ領域を節約することができる。

## 【 0 0 6 4 】

本発明の特徴として、キーテーブルのエントリは固定長だが、それぞれのテーブルで使用される鍵の長さは暗号解析能力の向上に対応できるように可変長とし、キーテーブルのキーサイズ領域で指定される。マイクロプロセッサ 1 0 1 に固有の秘密鍵  $K_s$  は固定されているが、プログラムの暗号化、復号化に用いられる  $K_{code}$  の長さは鍵エントリの指定によって変えられるということである。可変長の鍵の位置を指定するため、鍵エントリ  $309-m$  には鍵エントリをポイントするフィールド  $309-m-4$  があり、キーオブジェクト 310 のアドレスを示している。

## 【 0 0 6 5 】

キーオブジェクト領域 310 には、実行コードの暗号化鍵  $K_{code}$  が、公開鍵アルゴリズムにより、マイクロプロセッサ 1 0 1 の公開鍵  $K_p$  で暗号化された形  $E_{K_p}[K_{code}]$  で格納されている。公開鍵アルゴリズムでデータを安全に暗号化するためには、大きな冗長度が必要なため、暗号化されたデータ長は元のデータ長より長くなる。ここでは、 $K_s$ 、 $K_p$  の長さを 1 0 2 4 b ビット、 $K_{code}$  の長さを 6 4 b ビットとし、パディングにより 2 5 6 ビットの長さとして、 $E[K_{code}]$  を 1 0 2 4 ビットの長さに暗号化してキーオブジェクト領域 310 に格納する。 $K_{code}$  が長く、1 0 2 4 ビットに格納できない場合は、複数の 1 0 2 4 ビットブロックに分割して格納する。

## 【 0 0 6 6 】

図 6 は、上記の流れをまとめたものである。プログラムカウンタ 501 は、論理アドレス空間 502 の上の暗号化されたコード領域 503 の上のアドレス  $Addr$  を指している。論理アドレス  $Addr$  は、命令 TLB 121 に読み込まれたページテーブル 307 に基づいて、物理アドレス  $Addr'$  に変換される。これと同時に、暗号化されたコード復号化鍵  $E[K_{code}]$  がキーテーブル 309 から取り出され、復号化機能 506 において CPU の持つ秘密鍵  $K_s$  で復号化されて、カレントコード復号化鍵記憶手段 507 に格納される。コード暗号のための共通鍵  $K_{code}$  は、プログラムベンダにより、マイクロプロセッサ 1 0 1 の公開鍵  $K_p$  で暗号化され、 $K_{code}$  で暗号化されたプログラムとともに供給されるので、マイクロプロセ

ッサ101の秘密鍵Ksを知らないユーザは、Kcodeを知ることはできない。

【0067】

プログラムベンダは、Kcodeでプログラムの実行コードを暗号化し出荷した後は、Kcodeを安全に保管し、第3者に秘密が洩れないように管理する。

【0068】

全体キーテーブル511と、全体ページテーブル512は、物理メモリ510に置かれ、それぞれの物理アドレスがキーテーブルレジスタ508、CR3レジスタ509によって指定されている。これらの全体テーブルの内容は、バスインタフェースユニット112を介して、必要な部分だけが命令TLB121にキャッシュされる。

【0069】

さて、命令TLB112によって変換された物理アドレスAddr'に対応する内容503がバスインタフェースユニット112によって読み出されると、このページは暗号化されているので、コード復号化ユニット212で復号化される。読み出しは前記キャッシュラインサイズ単位で行われ、ブロック単位で復号化された後、上述したインタリーブの逆処理が行なわれる。復号された結果はL1命令キャッシュ213に格納され、命令として実行される。

【0070】

その他、プログラムのロード手法、リロケーションなどについては、同一出願人による特許出願第2000-35898号に記載されている。

【0071】

このように暗号化されたコードを実行できることにより、本発明のマイクロプロセッサでは、コードを逆アセンブルしてプログラムの動作を解析することを、暗号化アルゴリズムとパラメータを適切に選びさえすれば暗号学的に不可能とすることができる。

【0072】

同様に、ユーザはコード暗号化鍵Kcodeの真の値を知ることができないので、暗号化されたプログラムの一部を改変してユーザがそのアプリケーションが扱うコンテンツの不正コピーを取るなどのユーザの意図に沿った改変も暗号学的に不



可能にすることができる。

【 0 0 7 3 】

次に、デバッグ機能の抑止について説明する。

【 0 0 7 4 】

命令 T L B 1 2 1 は、現在実行中のコードが保護されているかどうか（暗号化されているかどうか）を判定することができる。保護されたコードの実行中は、デバッグフラグやデバッグレジスタから暗号化プログラムの解析に侵入されることを防止するため、デバッグレジスタ機能と、ステップ実行機能という 2 つのデバッグ機能が禁止される。

【 0 0 7 5 】

デバッグレジスタ機能とは、プロセッサに備えられたデバッグレジスタに、メモリアクセス範囲や、実行、データとしての読み出し、書き込みなどのアクセス種別を設定しておくことにより、対応するメモリアクセスが発生したときに割り込みが発生する機能である。本実施形態においては、保護されたコードを実行している時は、デバッグレジスタに設定された内容は無視され、デバッグのために割り込みは発生しない。ただし、ページテーブルにデバッグのビットが設定されている場合を除く。ページテーブルのデバッグビットについては後述する。

【 0 0 7 6 】

非保護（平文）のコードを実行中は、プロセッサの E F L A G S レジスタのステップ実行ビットをセットすると、1 命令を実行する度に割り込みが発生するが、保護されたコードの実行中には、このビットも無視され、割り込みは発生しない。

【 0 0 7 7 】

本発明では、実行コードの暗号化による解析の防止に加えて、これらの機能により、デバッグレジスタやデバッグフラグによるプログラムの動的解析を防止することで、ユーザによるプログラムの解析を困難にしている。

【 0 0 7 8 】

次に、本発明のマイクロプロセッサの特徴である、マルチタスク環境下でのプログラム実行中断時における、コンテキストの暗号化と署名および検証について

説明する。

【 0 0 7 9 】

マルチタスク環境でプログラムの実行は、例外によりしばしば中断される。通常、実行が中断された時にはプロセッサが保持する状態は一旦メモリ上に保存され、後でそのプログラムの実行が再開される時に元の状態が復帰される。これにより、複数のプログラムによる処理を疑似的に並行して実行したり、割り込み処理を受け付けることが可能になっている。この中断時の状態情報は、コンテキスト情報と呼ばれる。コンテキスト情報にはアプリケーションが使用するレジスタ情報が含まれ、それに加えて、明示的には見えないがレジスタの情報が含まれる場合がある。

【 0 0 8 0 】

従来のプロセッサでは、あるプログラムの実行中に割り込みが発生すると、アプリケーションのレジスタ状態が保持されたままOSの実行コードに制御が移されるので、OSはそのプログラムのレジスタ状態を調べてどのような命令を実行していたかを推定したり、平文のまま保存されたコンテキスト情報を実行の中断中に改変することで、そのプログラムの実行再開後のプログラムの動作を変えることができた。

【 0 0 8 1 】

そこで、本発明では、保護されたコードの実行中に割り込みが発生した時に、その直前に実行していたコンテキストを暗号化して保存し、全てのアプリケーションレジスタを暗号化または初期化するとともに、コンテキスト情報にプロセッサによる署名を添付する。中断からの復帰時に署名を検証し、その署名が正しい署名かどうかをチェックする。署名の不整合を検出した場合には復帰を中止して、ユーザによる不正なコンテキスト情報の改変を防止することができる。このとき、暗号化の対象となるレジスタは、図7の701～720までのユーザレジスタである。

【 0 0 8 2 】

Pentium Proアーキテクチャでは、プロセスのコンテキスト情報のメモリ上への保存、復帰をハードウェア的に支援する機構がある。この状態を保存するため

の領域は T S S (task state segment) と呼ばれる。以下この機構に本発明を適用した例を説明するが、これは本発明の適用を Pentium Pro アーキテクチャに限定するものではなく、本発明は広く一般のプロセッサアーキテクチャに適用が可能である。

## 【 0 0 8 3 】

例外発生にともなうコンテキスト情報の保存は次の場合に起きる。例外が発生すると、I D T (Interrupt Descriptive Table) と呼ばれる例外処理を記述するテーブルの中から、割り込み原因に対応するエントリが読み出され、そこに記述された処理が実行される。エントリが T S S を示すものであるとき、指示された T S S に保存されていたコンテキスト情報がプロセッサに復帰される。一方、それまで実行されていたプロセスのコンテキスト情報は、そのときのタスクレジスタ 7 2 5 で指定された T S S 領域に保存される。

## 【 0 0 8 4 】

この自動的なコンテキスト保存機構を使えば、プログラムカウンタやスタックポインタを含めて全てのアプリケーション状態を保存し、復帰時に署名を検証することで改竄の有無を検出することができる。しかし、自動的なコンテキストの保存を行う場合、コンテキストの切替のために大きなオーバーヘッドが生じる他に、T S S を使わない割り込み処理ができないなどの問題が生じる。

## 【 0 0 8 5 】

割り込み処理のオーバーヘッドを小さくする、あるいは既存プログラムとの互換性を保つためには自動的なコンテキスト保存の機構を使わないことが望ましいが、この場合には、プログラムカウンタはスタック上に保存されてしまい、検証の対象とはできないため、悪意のある O S による改竄の対象ともなりうる。両者は目的に応じて使い分けられることが好ましい。そこで、本発明のマイクロプロセッサでは、保護された（暗号化された）実行コードに対しては、安全性を重視して、自動的なコンテキスト保存を採用する。自動的に保存するレジスタは必ずしも全てのレジスタでなくてもよい。

## 【 0 0 8 6 】

本実施例におけるコンテキスト保存と再開処理の特徴は次の 3 点である。

【 0 0 8 7 】

(1) コンテキストを生成したマイクロプロセッサ自体と、コンテキストを生成したプログラムの暗号化鍵K codeを知る者だけが、保存されたコンテキストの内容を復号化できる。

【 0 0 8 8 】

(2) あるコード暗号鍵Xで保護されたプログラムが中断され、そのコンテキストが保存された場合、その再開処理は非保護のプログラムや別のコード暗号化鍵Yで暗号化されたプログラムの再開に適用することはできない。すなわち、中断から復帰させるべきプログラムが、再開時に別のプログラムに入れ代えられることはない。

【 0 0 8 9 】

(3) 改変されたコンテキストの復帰を禁止する。すなわち、仮に保存されていたコンテキストが改変されていた場合、そのコンテキストは復帰されない。

【 0 0 9 0 】

上記の特徴(1)により、コンテキスト情報の安全性を維持すると同時に、プログラムベンダによるコンテキスト情報の解析は可能にしておくことができる。プログラムベンダがコンテキスト情報を解析する権利を保持していることは、ユーザの使用条件で発生した不具合の原因を解析し、プログラムの品質を維持するためにも重要だからである。

【 0 0 9 1 】

特徴(2)は、たとえば攻撃者が、プログラムAの実行によって生成されたコンテキストを、別の暗号プログラムBに適用して、コンテキストに保存された既知の状態からプログラムを再開することによって、プログラムBに含まれるデータやコードの秘密を解析したり動作を改変することを防ぐためのものである。この機能はまた、後述するデータ保護に際して、複数のアプリケーションが互いに独立して、それぞれ排他的に暗号化データを保持するための前提となる。

【 0 0 9 2 】

特徴(3)により、プログラムの再開時を利用したコンテキスト情報の改変を厳密に排除することができる。

## 【0093】

このような機能を持たせる理由は、単にプロセッサの秘密情報に基づいてコンテキスト情報を暗号化しただけでは、秘密情報を、攻撃者の意図に沿ったコンテキスト情報の改変から守ることはできても、コンテキストが無秩序に改変され、ランダムに誤りが生じた状態からプログラムが再開される可能性を排除することができないからである。

## 【0094】

以下、上記3つの特徴をそなえたコンテキスト保存と検証方法について詳細に説明する。

## 【0095】

## ＜コンテキスト保存処理＞

図8は、第1実施形態のコンテキスト保存形式を概念的に示した図である。保護されたプログラムの実行中にハードウェア、またはソフトウェアが原因の割り込みが発生したとする。割り込みに対応するIDTエントリがTSSを示すものであれば、それまでのプログラムの実行処理が暗号化され、コンテキスト情報として（そのTSSではなく）カレントのタスクレジスタ725が示すTSSに保存される。そして、IDTエントリが示すTSSに保存された実行状態がプロセッサに復帰される。IDTのエントリがTSSを示していない場合、カレントのレジスタの暗号化または初期化のみが行なわれ、TSSの保存は行なわれない。当然そのプログラムの再開は不可能となる。ただし、OSの動作の継続のため、レジスタの暗号化または初期化の対象から、フラグレジスタの一部タスクレジスタを含むシステムレジスタは除外する。

## 【0096】

図8に示すコンテキストは、実際には、その内容がインタリーブされ、ブロック毎に暗号化されてメモリ上に保存される。まず保存される情報の項目について説明する。先頭には各特権モードに対応したスタックポインタとユーザレジスタ802～825があり、その次に、TSSのサイズと暗号化の有無を示す1ワード826が置かれる。これはプロセッサが保存されたTSSが暗号化されたものであるかどうかを示すもので、TSSが暗号化された場合でも、この領域は暗号

化されずに、平文のまま保存される。続いてデータの保護のために追加されたデータ暗号化制御レジスタ（CY0～CY3）の領域827～830と、サイズをブロック長に合わせるためのパディング831が置かれる。最後にコンテキストを暗号化した鍵 $K_r$ を実行コードの暗号化 $K_{code}$ で共通鍵アルゴリズムにより暗号化した値 $E_{K_{code}}[K_r]$ 832、コンテキストを暗号化した鍵 $K_r$ をプロセッサの公開鍵 $K_p$ で暗号化した値 $E_{K_p}[K_r]$ 833、そして、これら全体に対するプロセッサの秘密鍵 $K_s$ による署名 $S_{K_s}[message]$ 834が置かれる。また、タスク間の呼出し関係を保持する前回のタスクとのリンク領域801は、OSによるタスクスケジューリングを可能にするため平文のまま保存される。

## 【0097】

これら実行コードの暗号化や署名生成は、図2に示す例外処理部131の中のコンテキスト情報暗号化／復号化ユニット254によって行なわれ、実行コードの処理対象であるデータの暗号化とは独立した機能に基づく。コンテキスト情報がTSSに保存される際には、別途のデータ暗号化機能によりTSSのアドレスになんらかの暗号化が指定されていたとしても、その指定は無視され、コンテキストの暗号化が行なわれたそのままの状態では保存される。データ暗号化機能の暗号化属性は、それぞれの保護された（暗号化された）プログラムに固有のものであるため、あるプログラムの再開をその機能に依存することができないためである。

## 【0098】

コンテキストの暗号化にあたっては、まず平文のまま記録されるTSSサイズ領域826のワードが、値0に置き換えられる。そして、図5と関連して説明したのと同様のインタリーブが施されて、コンテキストが暗号化される。このとき、パディング831は暗号化ブロックサイズに合わせてインタリーブが適切に行えるサイズに設定する。

## 【0099】

ここで、レジスタ値をプロセッサの公開鍵 $K_p$ またはコードの暗号化鍵 $K_{code}$ で直接暗号化しないのは、プログラムベンダとプロセッサの両者による暗号化されたコンテキストの解読を可能にし、同時にユーザによるコンテキストの復号化

を禁止するためである。

#### 【0100】

プログラムベンダはコードの暗号化鍵  $K_{code}$  を知っているので、 $K_{code}$  を使用して  $E_{K_{code}}[K_r]_{832}$  を復号化して、コンテキストの暗号化鍵  $K_r$  を取り出すことができる。また、マイクロプロセッサ 101 は、内部に持つ秘密鍵  $K_s$  で  $E_{K_P}[K_r]_{833}$  を復号化して、 $K_r$  を取り出すことができる。すなわち、プログラムベンダは、ユーザのマイクロプロセッサの秘密鍵を知ることなく、コンテキスト情報を復号化して不具合の解析を行うことができ、マイクロプロセッサ 101 自体は、内部に持つ秘密鍵でコンテキスト情報を復号化して実行を再開することができる。いずれの鍵も持たないユーザは、保存されたコンテキスト情報を復号化することはできない。また、コンテキスト情報と、上記  $E_{K_{code}}[K_r]$ 、 $E_{K_P}[K_r]$  に対する署名  $S_{K_S}[\text{message}]$  を、マイクロプロセッサ 101 の秘密鍵  $K_s$  を知らないユーザが偽造することもできない。

#### 【0101】

プログラムベンダとマイクロプロセッサによる互いに独立したコンテキスト情報の復号化を可能とするために、直接  $K_{code}$  でコンテキスト情報を暗号化する方法も考えられる。しかし、レジスタの状態が既知の場合、コードの暗号化鍵  $K_{code}$  に対して既知平文攻撃が行なわれるおそれがある。すなわち、データを暗号化するための鍵の値が固定されていると、次のような問題がある。ユーザのデータ入力を読み込んで、それを一時的に作業用メモリに暗号化して書き込むというプログラムを実行していたとする。暗号化されて書き込まれるデータは、メモリ上を観察していればわかるので、ユーザは入力の値を変えて何度も入力を繰り返し、それに対応する暗号化されたデータを入手することができる。これは、暗号解析の理論における「選択平文攻撃」が可能であることを意味する。

#### 【0102】

共通鍵暗号アルゴリズムにとって既知平文攻撃は致命的ではないとはいえ、これを回避するに越したことはない。そこで、例外処理部 131 の乱数発生ユニット 252 により、コンテキストの保存の都度、乱数  $K_r$  を生成し、コンテキスト情報暗号／復号化ユニット 254 に供給する。コンテキスト情報暗号／復号化ユ

ニット254は、乱数 $K_r$ を用いて、共通鍵アルゴリズムによりコンテキストを暗号化する。そして、乱数 $K_r$ をコード暗号化鍵 $K_{code}$ で同じく共通鍵アルゴリズムにより暗号化した値 $E[K_r]$  832として添付する。プロセッサの公開鍵 $K_p$ による $K_r$ の暗号化 $E_{K_p}[K_r]$  833は、公開鍵アルゴリズムによる。

## 【0103】

ここで、乱数は、乱数発生機構252によって生成される。暗号化されているのがプログラムの場合には、通常はプログラムコードには変化がなく、動作の解析をしない限り、対応する平文のコードが不正に入手されることはない。この場合、暗号を破るには「暗号文単独攻撃」を行う必要があり、暗号化鍵の探索は非常に困難である。ところが、ユーザによって入力されたデータを暗号化してメモリに格納する場合には、ユーザが入力データを自由に選択できる。このため、暗号化鍵に対して「選択平文攻撃」という、「暗号文単独攻撃」と比べてはるかに効率的な攻撃が可能となってしまう。選択平文攻撃に対しては、保護されるべき平文に「salt（塩）」と呼ばれる乱数を追加して探索空間を大きくするという対策が可能である。しかし、すべてのデータにsaltの乱数値を組み込んだ形でメモリに保存することを、アプリケーションプログラミングのレベルで実装するのは非常に繁雑であり、プログラミング効率と性能の低下を招くことになる。

## 【0104】

そこで、乱数発生機構253は、コンテキストの保存の都度、それを暗号化するための乱数（暗号化鍵）を発生する。暗号化鍵を任意に選択できることにより、プロセス間、プロセス—デバイス間の安全な通信を高速化できるという効果も生じる。メモリアクセス時にハードウェアによってデータを暗号化する速度は、ソフトウェアによって暗号化する速度に比べて一般にはるかに速いためである。これに反し、データ領域の暗号化鍵の値が、予め決められた値、例えば実行コードの暗号化鍵と同一のものに制限されていると、別の暗号化鍵で暗号化された他のプログラムや、デバイスとの暗号化されたデータの共有にプロセッサのデータ暗号化機能を使うことができず、プロセッサに設けられたハードウェア暗号化機能の高速性を生かすことができないからである。

## 【0105】



なお、署名 834 の生成および再開時に行われる暗号化された乱数  $E[K_r]$  832 の復号化は、マイクロプロセッサ 101 だけが行えるという条件を満たしていれば、どのようなアルゴリズムと秘密情報に基づいてもよい。上の例では、ともにマイクロプロセッサ 101 に固有の秘密鍵  $K_s$ （これはコード暗号化鍵  $K_{code}$  の復号にも用いられる）を使っているが、それぞれ別の値を使ってもかまわない。

## 【0106】

また、保存されたコンテキストには暗号化の有無を示すフラグがあり、暗号化されたコンテキスト情報と暗号化されないコンテキスト情報が用途に応じて共存できる。TSS のサイズと暗号化の有無を示すフラグは平文で格納されているので、過去のプログラムとの互換性を維持するのが容易である。

## 【0107】

## ＜中断されたプログラムの再開処理＞

コンテキストを復帰させてプロセスを再開させる時、OS は保存された TSS を示す TSS デスクリプタへのジャンプまたは call 命令を発行する。

## 【0108】

図 2 のブロック図に戻ると、例外処理部 131 のコード暗号化鍵・署名検証ユニット 257 は、最初に署名  $S_{K_s}[\text{message}]$  834 をプロセッサの秘密鍵  $K_p$  を使って検証し、検証結果を例外処理ユニット 255 に送る。検証結果が失敗である場合は、例外処理ユニット 255 はコンテキストの再開を中止し、例外を発生させる。この検証により、コンテキスト情報が確かに秘密鍵を持つ正当なマイクロプロセッサ 101 により生成されたものであって、改竄を受けていないことが確認できる。

## 【0109】

署名の検証に成功すると、コンテキスト情報暗号／復号化ユニット 254 は、コンテキスト暗号化鍵  $E_{K_p}[K_r]$  833 を秘密鍵  $K_s$  で復号化して、乱数  $K_r$  を取り出す。取り出した  $K_r$  によりコンテキストを復号化する。一方、プログラムカウンタ (EIP) 809 に対応するコード復号化鍵  $K_{code}$  がページテーブルバッファ 230 から取り出され、コード暗号化／復号化鍵領域 251 に送られる

。コンテキスト情報暗号／復号化ユニット254は、コード復号化鍵 $K_{code}$ で $E_{K_{code}}[K_r]832$ を復号化し、その結果をコード暗号化鍵・署名検証ユニット257に送る。コード暗号化鍵・署名検証ユニット257は、 $E_{K_{code}}[K_r]832$ の復号結果が、マイクロプロセッサの秘密鍵 $K_s$ による復号化結果と一致するかどうかを検証する。この検証により、このコンテキスト情報が秘密鍵 $K_{code}$ で暗号化されたコードの実行により生成されたものであることが確認できる。

#### 【0110】

もしコンテキスト情報に対して、このコード暗号化鍵に関する検証を行わない場合には、ユーザが適当な秘密鍵 $K_a$ で暗号化したコードをつくり、それを実行して得たコンテキスト情報を、別の秘密鍵 $K_b$ で暗号化したコードに適用する攻撃が可能となってしまう。上記の検証はこの攻撃を排除し、保護されたコードのコンテキスト情報の安全性を保証している。

#### 【0111】

この目的は、コンテキスト情報に秘密のコード暗号化鍵 $K_{code}$ を追加することによっても達成できるが、本発明では、コンテキスト情報を暗号化する秘密の乱数 $K_r$ を、プログラムベンダの選んだコード暗号鍵 $K_{code}$ で暗号化した値 $E_{K_{code}}[K_r]$ を検証に使うことで、コンテキスト情報の保存に必要なメモリの量を削減し、コンテキスト切替の高速化とメモリの節約の効果を達成している。これはまた、プログラム作成者へのコンテキスト情報のフィードバックをも可能にする。

#### 【0112】

さて、コード暗号化鍵・署名検証ユニット257により、コード暗号化鍵の検証と署名の検証に成功すると、コンテキストがレジスタファイル253に復帰され、プログラムカウンタの値も復帰されるので、制御がコンテキストを生成した実行中断時のアドレスに戻される。

#### 【0113】

いずれかの検証に失敗して、例外処理ユニット255が例外を発生させた場合は、例外の発生アドレスはジャンプまたはcall命令が発行されたアドレスを示す。また、IDTテーブルの割り込み原因の領域にTSSの不正を示す値が格納さ

れ、割り込み原因となるアドレスを格納するレジスタにジャンプ先のTSSのアドレスが格納される。これにより、OSはコンテキスト切替失敗の原因を知ることができる。なお、再開処理を高速化するため、コンテキスト情報暗号／復号化ユニット254により復号された実行状態のレジスタファイル253へ供給と、検証ユニット257による検証処理を並行して行ない、検証に失敗した時は以降の処理をとりやめる構成としてもよい。

## 【0114】

乱数を介したこの暗号化方式の安全性は、使用する乱数系列の予測不可能性に依存するが、予測が困難な乱数をハードウェアによって生成する方法は例えば、小野寺他による日本国特許番号第2980976号に記載されている。

## 【0115】

プログラムベンダによるコンテキスト情報の解析は、ユーザの使用条件で発生したプログラムの不具合の原因を解析し、プログラムの品質を向上させる上で重要である。本実施例ではその点を鑑み、コンテキストの安全性とプログラムベンダによるコンテキスト情報の解析可能性の両立させる方式を説明しているが、この方式により、コンテキスト保存のオーバーヘッドが増えることも事実である。

## 【0116】

また、マイクロプロセッサの署名によるコンテキスト情報の検証は、不正なコンテキスト情報で保護されたコードを、任意に選んだ値と暗号化鍵の組み合わせにより実行されることを防止しているが、この追加もやはりオーバーヘッドを増やしている。

## 【0117】

プログラムベンダによるコード解析の必要性や、不正なコンテキスト情報によるプログラム再開を排除する機構が必要ない場合は、コードの暗号化鍵を特定する情報を含むコンテキスト情報をプロセッサの持つ秘密鍵で直接暗号化してもよい。これだけでコンテキストの意図的な改竄は暗号学的に不可能、かつ暗号化鍵の異なるプログラムに、コンテキスト情報が適用されることを防ぐことができる。

## 【0118】

ここで、コンテキスト保存の形式について説明を追加しておく。動作との関係については後述する。

#### 【 0 1 1 9 】

図 8 において、8 2 5 - 1 に示す R ビットはコンテキストが再開可能かどうかを示すビットである。このビットが 1 にセットされている場合、上記復帰手順でコンテキストに保存された状態を復帰して実行を再開できるが、値が 0 の場合には、再開することはできない。これは、暗号化プログラムの実行中に不正が検出されたコンテキストの再開を防ぐことで、再開可能なコンテキストを正しい状態のものに限定できるという効果を有する。

#### 【 0 1 2 0 】

8 2 5 - 2 に示す U ビットは T S S がユーザ T S S かシステム T S S かを示すフラグである。このビットが 0 の場合は保存された T S S はシステム T S S であり、このビットが 1 にセットされている場合は保存された T S S はユーザ T S S である。上の例で説明した例外エントリからの特権の変更を伴うタスク切り替えや、タスクゲートの呼出しを通して保存、復帰される T S S はシステム T S S である。一方、特権の変更を伴わないユーザによるサブルーチン呼出し命令、明示的なコンテキスト保存命令の実行によって保存される T S S はユーザ T S S である。システム T S S とユーザ T S S の違いは、T S S の復帰時に現在実行中のプログラムの T S S 保存場所を示すタスクレジスタが更新されるかされないかにある。システム T S S の復帰では、現在実行中のプログラムのタスクレジスタが、新しく復帰される T S S の前回のタスクとのリンク領域 8 0 1 に保存され、新しい T S S のセグメントセレクトがタスクレジスタに読み込まれる。一方、ユーザ T S S の復帰では、タスクレジスタの値の更新は行われず、ユーザ T S S はプログラムのレジスタ状態の保存、復帰のみを目的としており、特権モードの変更は伴わない。

#### 【 0 1 2 1 】

### < データ保護 >

次に、実行コードの演算処理対象であるデータの保護について説明する。

#### 【 0 1 2 2 】

第1実施形態では、データを保護するための暗号化属性は、マイクロプロセッサ101の内部に設けられたCY0～CY3の4つのレジスタに定義される。図7に示す領域717～720がこれに該当する。図7では、CY0～CY2については詳細を省略し、CY3のみを詳細を示している。

#### 【0123】

暗号化属性の各要素を、CY3レジスタ717を例にとって説明する。暗号化される領域の先頭を示す論理アドレスの上位ビットが、ベースアドレス717-1に指定される。領域のサイズはサイズ領域717-4に指定される。サイズはキャッシュライン単位で指定されるため、下位のビットには無効部分がある。データの暗号化鍵は717-5に指定される。共通鍵アルゴリズムを使用しているので、復号化鍵にも717-5を使用する。暗号化鍵の値に0が指定された時、そのレジスタが示す領域は暗号化されていないことを示す。領域の指定はCY0が優先され、以下CY1～CY3の順番で優先される。例えばCY0とCY1の指定する領域が重なった場合、その領域ではCY0の属性が優先される。また、処理対象のデータではなく、実行コードとしてのメモリアクセスではページテーブルの定義が優先される。

#### 【0124】

717-4はデバッグビットであり、デバッグ状態でのデータ操作を暗号化状態で行うか、平文状態で行うかを選択する。デバッグビットについての詳細は後述する。

#### 【0125】

図10は、実行コードの処理対象であるデータの暗号化、復号化の流れを示す。ここでは、コードが保護された状態、すなわち暗号化された状態で実行されている時のみ、データの保護が行われる。ただし、後述するデバッグ状態でコードが実行されている場合は除外する。コードが保護されている時、暗号化属性レジスタCY0～3の内容は、図2に示すレジスタファイル253からデータTLB141内にあるデータ暗号化キーテーブル236に読み込まれている。

#### 【0126】

ある命令が論理アドレスAddrにデータを書き込む時、データTLB141は、

論理アドレスAddrがCY0～3の範囲に含まれているかどうかを、データ暗号化キーテーブル236（図2参照）を調べることにより判断する。判断の結果、暗号化属性が指定されていれば、データTLB141は、L1データキャッシュ114からメモリへの対応キャッシュラインのメモリ書き出しの際に、メモリ内容を指定された暗号化鍵で暗号化するように、データ暗号化ユニット220に指示する。

## 【0127】

読み込みの場合も同様に、対象アドレスが暗号化属性を持っていれば、対応するL1データキャッシュ218のキャッシュラインの読み込みの際に、データ復号化ユニット219に対して、指定された暗号化鍵で復号化するように指示する。

## 【0128】

第1実施形態では、データ暗号化のためのデータ暗号化属性をすべて、マイクロプロセッサ101内部のレジスタに置き、実行の中断時にレジスタの内容をコンテキスト情報として安全な形でマイクロプロセッサ外部のメモリ（たとえば図2のメインメモリ281）に保存することにより、データ暗号化属性を、OSの特権も含めた不正な書き換えから保護している。

## 【0129】

データの暗号化、復号化は、コンテキストの暗号化と関連して先に説明したインタリーブを受けたキャッシュライン単位で行われる。このため、L1キャッシュ114上のデータを1ビット書き換えただけでも、メモリ上ではキャッシュライン内の他のビットが書き換えられることになる。データの読み書きの実行はキャッシュライン単位でまとめて行われるため、オーバーヘッドの増大はさほど大きくはないが、暗号化されたメモリ領域に対する読み書きは、キャッシュラインサイズ以下の単位では行えないことに注意が必要である。

## 【0130】

## &lt;エントリゲート&gt;

本発明では、保護されていないコードから、保護されたコードに制御が移行できるのは、次の2つの場合に限られる。

## 【0131】

(1) 再開アドレスと一致するコード暗号化鍵で暗号化された（乱数を持つ）コンテキストが再開される場合；

(2) 連続したコードの実行またはjump, call命令などにより、保護されないコードから、保護されたコードのエントリゲート命令（EGATE命令）に制御が移る場合。

## 【0132】

この限定は、攻撃者がコードを任意の場所から実行することによって、コード断片の情報を得ることを防ぐためのものである。(1)の手順については、コンテキストの復帰と関連してすでに説明した。すなわち、中断前に実行していたコードのコード暗号化鍵と一致するコンテキスト情報が含まれていること、およびマイクロプロセッサ101が与えた正しい署名が付加されていることが検証された場合に、保護されたコードの実行制御に移る。

## 【0133】

(2)の手法は、保護されないコードから保護されたコードへと制御を移す場合には、制御の最初にエントリゲート（egate）命令と呼ぶ特殊な命令を実行しなければ、保護コードの実行に移行できないとする処理である。

## 【0134】

図9は、エントリゲート命令に基づく保護ドメインの切り替え手順を示す。マイクロプロセッサ101は、例外処理部131のカレントコード暗号化鍵記憶手段251（図2参照）に、現在実行中のコードの暗号化鍵を保持している。まず、ステップ601で、命令の実行にともなってこの鍵の値が変更されたかどうかを判断する。鍵の値の変更が検出された場合（ステップ601でNO）、ステップ602に進み、変更に伴って実行されている命令がエントリゲート（egate）命令であるかどうかを調べる。エントリゲート命令であるということは、それが適正な命令であり、変更されたコードに制御を移行してもかまわないことを意味する。したがって、ステップ602でエントリゲート命令であると判定された場合は（ステップ602でYES）、その命令を実行する。

## 【0135】

ステップ602でエントリゲート命令でないと判断された場合は（ステップ602でNO）、割り込まれた命令が不適正な命令であることを意味する。この場合、ステップ603に進み、直前に実行されていた命令が暗号化（保護）されたものかどうかを判断する。保護されていない命令であれば、そのまま例外処理を発生させることができるが、保護されている命令である場合、一応その命令の保護を守りつつ例外処理しなければならないからである。

## 【0136】

したがって、ステップ603で保護されていない命令であると判断された場合は（ステップ603でNO）、そのまま例外処理を行い、保護された命令であると判断された場合には（ステップ604でYES）、保護状態を守ったまま、例外処理を発生させる。

## 【0137】

このような制御移行の制限により、平文コードから、エントリゲート命令がおかれた場所以外のコードへ、直接制御を移すことが禁止される。コンテキストを復帰させるということは、すでにそのプログラムがエントリゲートを通じて一度実行された状態を復帰させることである。したがって、保護されたプログラムを実行するには、必ずエントリゲートを通らなければならないことになる。プログラム中でエントリゲートを置く場所を最低限に押さえることにより、さまざまなアドレスからプログラムを実行してプログラムの構造を推定する攻撃を防ぐ効果がある。

## 【0138】

さらに、このエントリゲートでは、データの保護属性レジスタの初期化を行う。エントリゲートを実行すると、図7に示す保護レジスタCY0～CY3（717～720）の鍵領域（CY3では領域717-5）に、乱数K<sub>r</sub>がロードされる。暗号化対象先頭アドレスを0、サイズをメモリの上限までに設定し、論理アドレス空間のすべてを暗号化対象に設定する。実行コードにデバッグ属性が設定されていないければ、デバッグビット（CY3では717-3）は非デバッグとする。

## 【0139】



つまり、暗号化コードの実行開始時点では、すべてのメモリアクセスはエントリゲート実行時に決定された乱数  $K_r$  で暗号化されることになる。また、前述の通り実行コードの暗号化制御はページテーブルの定義が優先される。この乱数  $K_r$  はコンテキストの暗号化に使われる乱数とは独立に生成される。

#### 【0140】

この機構により、新たに実行される保護されたプログラムは、必ず全てのメモリアクセス開始時にランダムに決定された鍵によって暗号化される設定となる。

#### 【0141】

もちろんこのままではメモリ領域の全体が暗号化されたままなので、メモリを通じてシステムコールのパラメータを渡したり、他のプログラムとのデータ交換をすることができない。そこで、プログラムは順次必要なメモリ領域を平文としてアクセスできるように保護属性レジスタを設定して、自身の処理環境を整え処理を進める。優先度の低いレジスタ  $CY3$  は最初の乱数で暗号化される設定のままとしておき、その他のレジスタに平文アクセスの設定として暗号化鍵  $0$  を設定すれば、必要以外の領域を平文としてアクセスし、暗号化して秘密にするべきデータを誤って平文の領域に書き出してしまう危険を少なくすることができる。

#### 【0142】

保護属性レジスタ以外のレジスタの内容は、エントリゲートにおける初期化でも暗号化せずに、スタックやパラメータの場所を指定するためのポインタを格納しておくことができる。ただし、レジスタに不正な値を設定してエントリゲートを呼び出すことにより、プログラムの秘密が盗まれることがないように、エントリゲートを通して実行されるプログラムの処理には注意を払う必要がある。

#### 【0143】

プログラミング的には制約され、効率が悪くなるが、安全性を重視すればエントリゲートでは保護属性レジスタ以外の汎用レジスタも含めて、フラグ、プログラムカウンタ以外のすべてのレジスタを初期化する構成としてもよい。この場合でもスタックなどのパラメータはプログラムカウンタの相対アドレス、または絶対アドレスで指定したメモリ領域を通じて受け渡すことが可能である。ただし、ここでもコンテキスト保存の場合と同様、OSの動作の継続のため、レジスタの

暗号化または初期化の対象から、フラグレジスタの一部タスクレジスタを含むシステムレジスタは除外する。

#### 【 0 1 4 4 】

このように、第 1 実施形態にかかるマイクロプロセッサ 1 0 1 では、平文状態のプログラムから、保護されたプログラムへと制御が移る際に、最初に実行する命令をエントリゲート命令に制限し、エントリゲート命令の実行によってデータ暗号化属性レジスタを含むレジスタを初期化することで、保護された実行コードの断片的な実行を防ぎ、特にデータ保護状態の不正な設定を防いでいる。

#### 【 0 1 4 5 】

次に保護されたプログラムの実行制御について説明する。始めに保護ドメイン内に閉じた呼び出しおよび分岐について説明する。保護ドメイン内の呼び出しは通常のプログラムと全く同一である。図 1 1 に、保護ドメイン内の呼び出しおよび分岐の概念を示す。

#### 【 0 1 4 6 】

保護ドメインのコード 1 1 0 1 の実行は、保護ドメイン外のスレッド 1 1 2 1 が、保護ドメインの egate（エントリゲート）命令へと分岐することにより開始される。egate 命令の実行によってすべてのレジスタが初期化され、その後、プログラムの実行によって順次データ保護属性が設定される。jmp xxx 命令により、保護ドメイン内の分岐先 xxx 1 1 1 1 に制御が移り（処理 1 3 2 2）、アドレス ppp 1 1 1 2 にある call yyy 命令が実行される（処理 1 1 2 3）。スタックメモリ 1 1 0 2 に呼び出し元のアドレス ppp 1 1 1 2 がプッシュされ、呼出先 yyy 1 1 1 3 へと制御が移る。呼び出し先での処理が完了して、ret 命令が実行されると、スタックの戻り番地 ppp 1 1 1 2 へと制御が移る。実行コードの暗号化鍵が同一である間は実行制御に制限はない。

#### 【 0 1 4 7 】

次に保護ドメインから非保護ドメインへの呼び出しおよび分岐について説明する。この制御の移行には、保護ドメインから非保護ドメインへのプログラム作成者の意図しない移行を避けるためと、データ保護状態を保護するために、特殊な命令の実行および以下に説明するユーザ T S S の操作を行う。

## 【 0 1 4 8 】

図 1 2 は、保護ドメインから非保護ドメインへの呼び出しおよび分岐動作の概念図である。それぞれのドメインに、保護ドメインの実行コード 1 2 0 1 と、非保護ドメインの実行コード 1 2 0 2 が置かれている。また、ユーザ T S S 領域 1 2 0 3 と、非保護ドメインとのパラメータ受渡し領域 1 2 0 4 が設けられている。

## 【 0 1 4 9 】

実行は、スレッド 1 2 2 1 が `egate` 命令を実行することに始まる。保護ドメインのプログラムは、非保護ドメインのコードを呼び出す前に、予め定められたパラメータ領域 1 2 0 4 にユーザ T S S 領域 1 2 0 3 のアドレスを保存しておく。そして `ecall` 命令を実行して非保護ドメインのコードが呼び出される。`ecall` 命令は 2 つのオペランドをとる。一つは呼出先のアドレスであり、もう一つは実行状態の保存先である。`ecall` 命令は、呼出時のレジスタ状態（正確にはプログラムカウンタは `ecall` 発行後の状態）をオペランド `u T S S` で指定した領域に、これまでに説明した暗号化 T S S と同じ形式で保存する。以下、この領域をユーザ T S S と呼ぶ。

## 【 0 1 5 0 】

ユーザ T S S とシステム T S S との違いは、図 8 に示すユーザレジスタにおいて、T S S 上の領域 8 2 5 - 2 に U フラグがセットされていることである。動作の違いについては、後述する。ユーザ T S S のメモリへの保存においても、システム T S S へのコンテキスト情報の保存と同様、ユーザが保護レジスタ C Y 0 ~ C Y 3 に定義したデータ保護属性は適用されない。

## 【 0 1 5 1 】

呼出先の非保護ドメインのコードでは、`ecall` 命令の実行によりレジスタは初期化されているので、パラメータの受渡しができない。このため、予め定められたアドレス `param 1 2 0 4` からパラメータを取得し、必要な処理を行なう。非保護ドメインの中ではプログラミングに制限はない。図 1 2 の例では、サブルーチン `qqq 1 2 1 3` を呼び出している（矢印で示す処理 1 4 2 5）。例えば `exx` から `qqq` の呼出しまでの間に、スタックポインタの設定やパラメータのスタックへ

の複写を行うアダプタコードをおくことにより、サブルーチンqqqが有する呼び出しセマンティクスに、保護ドメインからの呼び出しを適応させることができる。処理結果は、メモリ上のパラメータ領域1204を通じて、呼出元へ送られる（処理1226）。サブルーチンの処理が完了すると、呼出元の保護ドメインへ制御を戻すため、sret命令が発行される（1227）。

## 【0152】

sret命令も、オペランドを持たないret命令とは異なり、ユーザTSSを指定するオペランドを一つとる。ここではパラメータ領域param1204に格納されたポインタを通じて、間接的にユーザTSS1203を復帰情報として指定している。sret命令によるユーザTSSの復帰が、システムTSSの復帰と大きく異なる点は、ユーザTSSを復帰してもタスクレジスタは全く影響を受けない点である。ユーザTSSのタスクリンクのフィールドは無視される。sret命令のオペランドに825-2Uフラグが0のシステムTSSが指定された場合は、復帰は失敗する。

## 【0153】

復帰の実行の際には、すでに説明した実行状態の復号化およびコード暗号化鍵と署名の検証が行なわれ、違反が検出された場合、秘密保護違反の例外が発生する。検証に成功すると、呼出元のecall命令の次の命令から実行が再開される。このアドレスはユーザTSSの中で暗号化され署名されているので、偽造することは暗号学的に不可能である。プログラムカウンタを除くすべてのレジスタ呼出前の状態に戻されてしまうので、保護ドメインのコードは、サブルーチンexxの実行結果をパラメータ領域1204から取得する。

## 【0154】

保護ドメインの処理が完了して制御を非保護ドメインに移す時は、ejmp命令が使われる。ejmp命令はecallとは異なり、状態の保存は行なわない。もしecall, ejmp以外のjmp, call命令によって保護ドメインから非保護ドメインへと制御が移された場合、秘密保護違反の例外が発生して、暗号化されたコンテキスト情報がシステムのTSS領域（タスクレジスタが示す領域）に保存される。なお、このときコンテキスト情報は再開不可にマークされる。なお、保護ドメイン内のア

ドレスをejmp命令の飛び先として指定しても違反とはならない。

【0155】

以上が保護ドメインから非保護ドメインの呼び出し手順とそれに使用される新たに追加された命令である。

【0156】

アプリケーションによるユーザTSSの復帰の際に、特権を持つOSがユーザTSSをすりかえる攻撃の可能性があるわけではない。しかし、そこで交換可能なTSS情報は、保護ドメインのコードの暗号化鍵が正しく管理されている限りは、必ずegateを通して実行を開始され、割り込みまたはユーザによる明示的な実行状態の保存によって保存されたコンテキスト情報だけである。このコンテキスト情報の入れ換えによって、アプリケーションの秘密が洩れる可能性は極めて小さく、かつ、攻撃者にとっては、どのようなコンテキスト情報の入れ替えを行えばアプリケーションの秘密を取得できるかを予測するのは極めて困難である。

【0157】

上述した保護ドメインから非保護ドメインの呼び出し手順は、呼出先で最初に実行される命令が被呼出し側のegate命令ならば、保護ドメインの間で制御を移す手順に適用することも可能である。

【0158】

この時、両者の間のパラメータの受渡し領域を、予め両者の間で認証鍵交換を行なうことで共有した暗号化鍵によって暗号化しておけば、保護ドメインの間の呼出を安全に行なうことができる。

【0159】

<スレッド間の制御>

次に保護されたプログラムの実行と、その同一プログラムの制御下におけるスレッドとの関係、および、そこに生ずる問題と解決手段について説明する。

【0160】

上記では、暗号化された（保護された）プログラム内部での実行制御自体は、通常のプログラムと全く変わらないことを説明した。このことは、ユーザプログラムのレベルで、スレッドの切替を行なうユーザスレッドについても当てはまる

ことであり、ユーザスレッドでは特にプログラムが保護されていることを意識する必要はない。問題は、カーネルレベルで実現されるスレッドが複数存在する時に、どのようにデータの保護情報を共有するかである。

#### 【0161】

始めに、ユーザスレッドとカーネルスレッドについて簡単に説明する。ユーザスレッドとは、OSによる介入なしに、ユーザプログラムすなわちアプリケーション内部で仮想的に複数のスレッドを並列実行することである。それぞれのスレッドは、独立したスタックとプログラムカウンタ値を持つ。一方、カーネルスレッドは、スレッドの切替をOSに依存するスレッド実装である。機能的な相違としては、ユーザスレッドでは、あるスレッドが自発的にスレッド切替えをするコードを呼び出さない限り、スレッド切替えは発生しないが、カーネルスレッドでは、割り当てた実行時間を超過したスレッドの実行は強制的に停止される。詳細は前述した文献「UNIXカーネル」を参照されたい。

#### 【0162】

内部機構的には、カーネルスレッドではコンテキスト情報がOSに処理され、スレッドの切替が行なわれる。一方、ユーザスレッドではアプリケーションがスレッド切替のコードを持ち、そこでレジスタのコンテキスト保存領域への保存が行なわれる。この動作を行なうのはアプリケーション自身なので、コードが保護されていても全く問題なくスレッド切替の動作が可能であり、かつそれぞれのスレッドはデータ保護状態も共有することができる。唯一注意すべき点は、スレッドコンテキストを保存する領域を暗号化していない場合、スレッドコンテキストをOSや他のアプリケーションに読みとられたり、改竄されるおそれがあることである。スレッドコンテキスト情報は暗号化された領域に保存して、他のプログラムによる読みとりや意図的な改竄を防止することが望ましい。

#### 【0163】

カーネルスレッドにおいては、スレッドの切替えはOSによって行なわれる。例えばあるスレッド処理がタイマ割り込みによって中断されたとき、そのスレッドが割り当てられた処理時間を使い尽くしている場合には、OSのスケジューリング機能が他のスレッドを実行することで、リアルタイム処理における応答性も

高められるなどの利点がある。また、複数のカーネルスレッドが動作する機能を備えないOSも、従来のUNIXなどのOSには存在するが、ほとんどすべてのUNIX実装が備える非同期的なシグナル配送機能は、コンテキスト切替の観点から、プロセスのメインのスレッドと独立に実行されるカーネルスレッドの一種と考えてよい。カーネルスレッドのサポートはほとんどのOSの実装に必要な機能である。

## 【 0 1 6 4 】

ところが、保護されたコードに、現在のカーネルスレッドの実装を適用しようとする問題が生じる。本発明のマイクロプロセッサでは、保存されたコンテキスト情報は暗号化により保護されており、安全性の観点から、その一部を読み出す手段は備えていない。したがって、従来の方法では、別々に起動されたカーネルスレッドがデータ保護状態を共有することはできない。カーネルスレッドはそれぞれが別々のTSSを持つが、コンテキスト情報を単純に別のTSS領域にコピーしたとしても、スタック領域も含めてすべて同一となってしまうため、それらは別々のスレッドとして独立に動作することはできなくなる。

## 【 0 1 6 5 】

そこで、本発明では、本発明のプロセッサに、プログラムによる、ある特定の時点でのコンテキスト情報の保存命令と、以下の手順によってカーネルスレッド間のデータ保護情報とを共有させ、カーネルレベルのマルチスレッド機能を可能にしている。

## 【 0 1 6 6 】

図14は、データ保護属性の共有に使われるスレッドテーブルを示す。スレッドIDは、そのカーネルスレッドのタスクレジスタの値である。スレッドID0は初期化のために特殊な役割を持つ。ユーザTSSは、そのスレッドが保護ドメイン外のサブルーチンを呼ぶ場合などに、状態を保存するためのユーザTSS領域へのポインタを格納する。パラメータは、OSからそれぞれのスレッドへのパラメータ渡しや、逆にスレッドから保護ドメイン外のサブルーチンを呼び出す際、パラメータの渡しに使われる。作業中フラグはテーブルおよびTSS0の書き替えの排他制御に使われる。

## 【 0 1 6 7 】

図 1 3 は、同一プログラム下でのデータ保護属性共有の手順を示すフローチャートである。OS は保護ドメインを最初に行うメインのスレッドの実行前に、図 1 4 に示すスレッドテーブルを初期化する。初期化はスレッド ID 0 のユーザ TSS フィールドに値 0 を書き込み、メインのスレッド TSS 1 のパラメータフィールドに、メインのスレッドの実行に必要なパラメータ、例えば引数などが格納されたパラメータブロックへのポインタを書き込むことによって行われる。

## 【 0 1 6 8 】

ステップ 1 3 0 1 で、カーネルスレッドは最初に `egate` を実行して保護ドメインのコードの実行を開始する。このとき、システムレジスタ以外のレジスタはすべて初期化される。次に、ステップ 1 3 0 2 で、コード中に予め埋め込まれたスレッドテーブルのアドレスからスレッドテーブルを読みとり、初期化用のスレッド ID 0 に対応するユーザ TSS フィールドの値を見ることによって、そのユーザ TSS フィールドで初期化が開始されているかどうかを判断する。ユーザ TSS フィールドの値が 0 であれば、まだ初期化されていない状態である。この場合、スレッドテーブルが未初期化であると判断して（ステップ 1 3 0 2 で NO）、ステップ 1 3 0 3 に進む。

## 【 0 1 6 9 】

ステップ 1 3 0 3 で、スレッド ID 0 の作業中フラグのフィールドに 1 を書き込み、初期化が開始されたことを示してから、データ暗号化属性を初期化する。データ暗号化属性の初期化は、実行される保護されたアプリケーションに依存する。ここではプログラムのロード時に確保されている固定アドレスの領域に、予めプログラムで決められた暗号化鍵を設定してもよいし、動的にメモリを確保して乱数による暗号化鍵を設定してもよい。データ暗号化属性の設定が完了すると、ステップ 1 3 0 4 に進み、たとえばスレッド 1（タスクレジスタがシステム TSS 1 を示している）の実行状態を、ユーザ TSS を保存する命令である `stctx` 命令により、`uTSS0` の領域に保存する。`stctx` 命令で保存される TSS は、図 8 に示す 8 2 5 - 2 領域の U フラグが 1 にセットされたユーザ TSS となる。`stctx` の次の命令が `uTSS0` の再開時に最初に実行される命令となる。このユ



ーザTSSの保存、復帰は、スレッドを識別するタスクレジスタにはいっさい影響しない（タスクレジスタは、割り込みによって中断されたスレッドに対応するシステムTSSが保存される場所を示す。）。

## 【0170】

次にステップ1308に進み、テーブルにセットしておいた作業中フラグをクリアし、同一プログラムにおける他のスレッド（たとえばスレッド2）の実行を再開させる。再開処理については、後述する。次にステップ1309で、ltr命令によってタスクレジスタを読み出し、スレッドIDを特定する。そして、ステップ1310で、スレッドテーブルのパラメータに指定された本来の処理を開始する。

## 【0171】

同一プログラムの下で、割り込みなどにより、他のスレッド2（タスクレジスタがシステムTSS2を示している）がこの保護ドメインのコードを実行する場合も、スレッドテーブルを初期化して、egate命令を実行するまでは同じである。2番目以降のスレッドは、すでに最初のスレッドによって初期化が行われているので、テーブルのスレッドID0に対応するユーザTSSが1にセットされている。したがって、ステップ1302で初期化済みと判断される（1302でYES）。ステップ1305で、スレッドテーブルが作業中にマークされているかどうかを判断する。作業中であれば（1305でYES）、スレッド2は休眠して（ステップ1306）、ステップ1305を繰り返し、テーブルに対する作業の完了により実行が再開されるのを待つ。

## 【0172】

ステップ1305でテーブルが作業中でなければ（1305でNO）、スレッドID0に対応するユーザTSS0を復帰し、uTSS0の保存直後の状態（ステップ1508）に制御が移る。このとき、uTSS0に保存されたデータ保護状態が復帰される。しかし、このままでは、これからスレッド1を実行するのかスレッド2を実行するのか区別がつかない。そこで、ステップ1309でタスクレジスタを読み出し、自分のスレッドのスレッドIDを特定して、パラメータを読み出し、ステップ1310で、必要な処理（例えばシグナルハンドラ）の実行

を行なう。u T S S の復帰ではタスクレジスタは更新されないので、スレッドは自己のスレッド I D を正しく取得できる。

### 【 0 1 7 3 】

一度実行を開始してデータ保護属性を共有したスレッドが、データ暗号化属性を変更する場合には、T S S 1 のスレッドを例にとれば、スレッド I D T S S 1 に対応するテーブルのパラメータフィールドに必要な情報が含まれたパラメータブロックを書き込み、u T S S 1 をユーザ T S S として指定して上述の ecall 命令を発行してからステップ 1 5 0 3 へ制御を移す。T S S 1 の作業中フィールドに作業中フラグを書き込み、他のスレッドに対して休眠を要求し、他のスレッドが休眠状態になるとデータ暗号化属性の変更を開始し、データ暗号化属性の更新が完了すると、u T S S 1 へと制御を戻す。

### 【 0 1 7 4 】

#### (第 2 実施形態)

第 1 実施形態では、データの暗号化鍵などのデータ暗号化属性を全てマイクロプロセッサ内部のレジスタファイル 2 5 3 に格納していた。この方法では、暗号化属性の種類が保護属性レジスタの数を越えて増えると、保護属性レジスタの入れ替えを余儀なくされ、プログラミングが繁雑になるとともに処理性能の低下につながる。一方、保護属性レジスタの数を増やすとコンテキスト切替の際にメモリに保存するデータの量が増え、やはり処理性能が低下してしまう。プロセス間通信の暗号化を、マイクロプロセッサの暗号化機能によって行う応用を考えれば、通信相手のプロセス毎に別の暗号化鍵を使うのが普通なので、暗号化属性の種類が多くなることは容易に想像できるので、それに対する対策が必要となる。

### 【 0 1 7 5 】

このような場合、一般に情報を直接マイクロプロセッサ内のレジスタに保持するのではなく、外部のメモリ（たとえばメインメモリ 2 8 1）上のテーブルに書き込んでおき、必要な部分だけを動的にプロセッサ内部に読み込む手法が知られている。しかし、秘密を守るという観点からは、この方法は危険である。なぜなら、メモリ上のテーブルが攻撃者によって別のデータ（たとえば解読可能な鍵）が格納されたテーブルにすりかえられてしまうおそれがあるからである。

## 【0176】

そこで、本発明の第2実施形態では、秘密保護に必要な情報の中で、動的に読み込まれる部分ごとに、あらかじめマイクロプロセッサによる署名を付加しておく。そして、この部分が外部のメモリテーブルからマイクロプロセッサ内部に読み込まれる時に、署名を検証する。この方式により、毎回のコンテキスト切替の際のオーバーヘッドを減少させると同時に、すり替えに対する安全性を保証している。

## 【0177】

図15は、本発明の第2実施形態にかかるマイクロプロセッサのレジスタファイル253の構成を示す。第2実施形態のマイクロプロセッサは、内部レジスタに、各プログラムから参照されるデータのための暗号化属性をあらかじめ格納する4個の暗号化属性レジスタCY0～CY3と、暗号化属性を特定するための情報を格納する60ワードのCT0～CT59を有する。CY0～CY3は、それぞれが10ワード、CT0～CT59は、それぞれ4ワードの大きさを持つ。

## 【0178】

CY0～CY3はレジスタに暗号鍵フィールドを持っているが、CT0～CT59は暗号化鍵フィールドを持たず、代わりに鍵へのオフセットアドレスを持っている。CT59のオフセットアドレスは1521-4で示されている。オフセットは鍵格納領域の先頭からアドレスを示し、鍵格納領域の先頭は暗号化属性レジスタCY0で定義される。

## 【0179】

暗号化属性情報特定レジスタCT59の開始アドレス1521-1は暗号化対象領域の先頭を指定し、サイズ1521-5はその長さを指定する。キーオフセット1521-6は、CY0で指定された鍵格納領域の先頭からの鍵へのオフセットを示し、そのアドレスは鍵の値とハッシュからなる鍵エントリが格納される。Eフラグ1521-4は、そのレジスタが有効かどうかを示し、Dフラグ1521-3はデバッグ状態かどうかを示している。暗号化アルゴリズムと鍵の長さを指定するフィールドはそれぞれ1021-2、1021-7である。このレジスタはSalt1021-8を有する。Saltについては、後述する。

## 【 0 1 8 0 】

図 1 6 は、マイクロプロセッサ内部の C Y 0 レジスタおよび暗号化属性情報特定レジスタ C T 0 と、これらのレジスタで定義される外部メモリ上の鍵格納領域の関係を示した図である。暗号化属性レジスタ C Y 0 によって定義されるメモリ上の鍵格納領域 1 6 0 1 には、各キーエントリ（鍵エントリ） 1 6 0 2、 1 6 0 3、…がある。各キーエントリは、鍵の値を示すフィールド 1 2 0 2 - 1、 1 2 0 3 - 1 と、この鍵値およびその他の情報についての書名 1 2 0 2 - 2、 1 2 0 3 - 2 を有する。

## 【 0 1 8 1 】

暗号化属性レジスタ C Y 0 ~ C Y 3 の初期化時には、暗号化属性情報特定レジスタ C T 0 ~ C T 5 9 の全てのレジスタの有効フィールド 1 5 2 1 - 4（図 1 5）がクリアされ、これらのレジスタで指定された暗号化機能が無効となっている。一方、C Y 0 ~ C Y 3 は有効で、論理アドレス空間のすべてを、初期化時に選択した乱数で暗号化するように設定されている。

## 【 0 1 8 2 】

C T 0 ~ C T 5 9 の暗号化機能を使う場合は、まず論理アドレス上の鍵の格納領域とその領域のためのマスターキーを選択し、それを C Y 0 に設定して、C Y 0 レジスタの暗号化機能を有効とする。マスターキーはマイクロプロセッサの例外処理部 1 3 1 の乱数生成機構 2 5 2 によって生成されたものでも、予めプログラムに埋め込まれた固定鍵でもよい。

## 【 0 1 8 3 】

次に、外部メモリ上の鍵格納領域 1 6 0 1 において、レジスタ C T 0 で使う鍵の格納場所（鍵エントリ） 1 6 0 3 を、暗号化属性レジスタ C Y 0 に設定したアドレス領域の中で決定して、C T 0 レジスタのオフセットフィールド 1 5 8 0 - 6（図 1 5）に書き込む。また、暗号化アルゴリズムと鍵の長さを選択して 1 5 8 0 - 2、 1 5 8 0 - 7 に書き込む。

## 【 0 1 8 4 】

そして、鍵の値を選択して、鍵値のフィールド 1 6 0 3 - 1 に書き込む。必要に応じて S a l t 1 5 2 1 - 8 に乱数を書き込んでから、有効ビット E 1 5 2 1

-4を1に設定する。

#### 【0185】

マイクロプロセッサは、保護テーブル管理ユニット233（図2参照）を有する。保護テーブル管理ユニット233は、予め定められたアルゴリズムによって、定められた署名の計算対象について、マイクロプロセッサの秘密鍵 $K_s$ に基づく署名を計算する。署名した値を、鍵の値（キーバリュー）に続く署名領域1603-2に書き込む。このとき、保護テーブル管理ユニット233は、鍵と署名の格納領域1603がCY0の範囲に入っているかどうか、およびCY0の暗号化機能は有効かどうかを確認する。鍵領域1603がCY0の範囲外である場合、あるいはCY0の暗号化鍵が0に設定されていて暗号化機能が有効でない場合には、後述するように、マイクロプロセッサの例外処理ユニット255が秘密鍵保護例外を発生させる。CT0~CT59の安全性はCY0の安全性に依存するので、CY0の暗号化鍵は単に乱数を選ぶだけでなく、使用する暗号化アルゴリズムについて安全なことを確認することが望ましい。

#### 【0186】

署名の計算対象には、暗号化鍵の値1603-1、CT0レジスタ1580の内容、および暗号化鍵の論理アドレスが含まれる。CT0レジスタ1580にはSaltが含まれているので、適切な頻度でSaltの値を変更すれば、攻撃者が過去に使われた暗号化された鍵エントリを再利用して、鍵エントリをすりかえることは、たとえCY0のその他のフィールドの値が同一であったとしてもきわめて困難である。もちろん、全く属性値の異なる鍵エントリを利用した場合の、鍵エントリのすり替えは暗号学的に不可能である。

#### 【0187】

データの読み込みの際には、論理アドレスと、暗号化属性レジスタCY0~CY3および暗号化属性情報特定レジスタCT0~CT59を比較し、暗号化属性を決定する。読み込み対象の論理アドレスがCT0で指定された領域ならば、オフセットの示す暗号化鍵を鍵キャッシュに読み込む。コード暗号化鍵・署名検証ユニット257は、読み込まれた鍵の値と、CT0の内容と、鍵のアドレスとから、マイクロプロセッサの公開鍵 $K_p$ によって署名を検証する。検証の結果がエ

ラーであれば、秘密保護例外が発生して処理を停止する。検証の結果が正しければ、データ TLB 141 の復号化ユニット 237 において、取り出された鍵でデータを復号化し、復号化されたデータがデータキャッシュ 218 に読み込まれる。

#### 【0188】

##### <再開不能ビット>

検証に失敗して秘密保護例外が発生した場合、コンテキストの再開可能ビットが0にクリアされて、以後そのコンテキストの再開は不可能となる。このような違反が発生するのは、何らかの攻撃者によって、秘密保護のためのデータが破壊されている可能性が高いため、これに続く実行を禁止して、秘密情報の流出を防いでいる。

#### 【0189】

なお、ここでは暗号化属性レジスタ CY0～CY2 に、暗号化対象領域の論理アドレスの先頭とサイズを設定する場合を例にとったが、この他にもページテーブルや、セグメントディスクリプタのようなテーブル形式を利用したメモリ暗号化属性の管理方式が考えられる。これらの方式にも本発明が適用されることは言うまでもない。

#### 【0190】

第2実施形態においては、データの暗号化属性の決定には必要のない、サイズの大きな鍵情報をプロセッサ外部のテーブルに格納して、プロセッサ内部のコンテキスト情報のサイズを低減すると同時に、外部のテーブルに格納される鍵情報の一つ一つにハッシュを付加し、読み込み時に検証することで、プロセッサ外部に格納される情報のすりかえによる攻撃を防止している。

#### 【0191】

ここで、署名は、マイクロプロセッサの暗号化属性情報特定レジスタの値を含むデータから生成されているため、暗号化属性情報特定レジスタが有する Salt の値を変えることにより、過去にプロセッサ外部のメモリに書き込んだ鍵エントリを無効化して、過去に使われた鍵エントリの流用を防止することができる。このような署名方法を使用せずに、たとえば、アドレス範囲属性と鍵の値だけか

ら署名を生成するならば、アドレス範囲さえ一致すれば過去に使われた鍵エントリを外部のメモリ上ですりかえる攻撃が可能となってしまう。その一方で、もし署名生成の対象に汎用レジスタが含まれていれば、汎用レジスタの値が変わる都度、署名を生成しなおさなければならず、性能の低下を招く。

## 【 0 1 9 2 】

本発明では、暗号化属性情報特定レジスタの属性値と、Salt値さえ書き換えなければ、署名を再計算する必要はない。実際の使用では、暗号化属性情報特定レジスタの設定頻度はそれほど高くはないと考えられるので、性能の低下を押さえつつ、鍵の値のすりかえを防止することができる。

## 【 0 1 9 3 】

第2実施形態にかかるマイクロプロセッサにおいては、すべての暗号化属性をマイクロプロセッサのレジスタ上に格納する場合と比較して、コンテキスト情報が著しく小さくなり、コンテキスト切替の際の暗号化や署名に必要な計算量や、コンテキストを保存するためのメモリトラフィックを減少させて、プロセッサの性能を向上させることができるのである。

## 【 0 1 9 4 】

署名をメモリ側でなく、レジスタの側においても同様にすり替えを防ぐことができるが、一般に非対称鍵方式を使った安全な署名には、大きなデータ長を必要とするので、レジスタの側にSalt値を置き、メモリ上に署名を置く方がコンテキスト情報を小さくする効果大きい。

## 【 0 1 9 5 】

本発明では、暗号化エントリを有効にする際に、対応するメモリ上の鍵格納領域1601が暗号化されていること（すなわちエントリに有効な鍵があること）を検証するので、暗号化鍵の保護をさらに確実にしている。

## 【 0 1 9 6 】

## &lt; ページテーブル形式 &gt;

また、データ保護のテーブル形式を、仮想記憶の管理に使われるページテーブルと同様のアドレス単位ごとに階層化された形式とし、各々のエントリ毎に署名を付加したものにすることも可能である。ページテーブル形式は理論的にはペー

ジ毎に異なる暗号化属性を与えることができる。さらに、マイクロプロセッサ内部に保持するデータは、ページテーブルの先頭アドレスとSaltだけでよく、コンテキスト情報の量を小さくすることができるという効果がある。

## 【0197】

混乱を避けるため以下、マイクロプロセッサ内部でデータ暗号化に使われるテーブルを、暗号化定義ページテーブルと呼ぶことにする。暗号化定義ページテーブルは、外部メモリの仮想記憶ページテーブルとは異なり、ユーザプログラムの論理メモリ空間上に置かれる。

## 【0198】

また、暗号化定義ページテーブルの各エントリに付与する署名が、単純にプロセッサの暗号化定義ページテーブルレジスタ情報およびエントリに含まれる情報だけから生成される場合、任意のエントリが署名を含んだ状態で交換されたとしても、それを検出することはできない。プロセッサ側に置かれる情報、すなわち暗号化定義ページテーブルレジスタとそれに含まれるSaltが全てのエントリについて共通となってしまうからである。この問題は、暗号化定義ページテーブルの各エントリの署名の計算領域に、エントリが置かれる論理アドレスを含めることによって（すなわち、エントリごとに異なる値が署名に含まれるようにすることによって）、解決される。

## 【0199】

なお、ページテーブルの初期化は、あらかじめキーバリュー（鍵値）などの属性をメモリのテーブルに書き込んでおき、そのアドレスをマイクロプロセッサ内部の暗号化定義ページテーブルレジスタに設定し、Saltの値を同じレジスタに書き込んでこのレジスタの有効フラグをセットする。有効フラグのセットにより、ページテーブルの各エントリに対する署名がマイクロプロセッサにより計算され、外部メモリ上に書き込まれる。階層化されている場合は階層を順次探索して計算が行なわれる。この処理は、エントリ数に比例した時間を要することに留意しなければならない。

## 【0200】

<デバッグについて>



実行コードと、その処理対象であるデータの双方が保護されるとしても、そのような機能の開発途上のデバッグ作業では、暗号化することなく平文でデバッグできることが開発効率上望ましい。

#### 【0201】

デバッグに関する設定は2箇所で行う。ひとつは、暗号化コードのページテーブルエントリに設けられた実行コードのデバッグビット307-j-D（図4参照）であり、もうひとつは、図15に示すように、データの暗号化鍵属性を指定するレジスタに設けられた暗号化制御ビット（CY3においては1517-3、CT59においては1521-3）である。後者の第1実施形態における対応ビットは、図7に示す暗号化属性レジスタCY3のビット717-3である。

#### 【0202】

ページテーブルエントリに設けられたデバッグビット307-j-Dがセットされている時、マイクロプロセッサの実行コード復号化ユニット212（図2参照）は迂回され、エントリに対応するメモリ上におかれた実行コードは、平文状態のまま解釈され、実行される。デバッグ状態でコードを実行している間は、プロセッサのデバッグ機能、たとえば第1実施形態と関連して述べたステップ実行機能やデバッグレジスタが有効化される。

#### 【0203】

ここで、ページテーブルのデバッグビットは何ら保護を受けていないが、悪意のあるユーザが、暗号化された実行プログラムのおかれているページテーブルエントリのデバッグビットをセットしたとしても、暗号化された実行プログラムが復号化されずに命令フェッチ／デコード機能214に取り込まれるだけで、正常実行はできないので、プログラムの秘密が洩れるおそれはない。

#### 【0204】

デバッグ実行中は、例外発生時のコンテキスト保存も平文のまま行われる。したがって、コンテキストを暗号化する乱数K<sub>r</sub>をコード暗号化鍵K<sub>code</sub>で暗号化した値E<sub>K<sub>code</sub></sub>[K<sub>r</sub>]833は、平文のK<sub>r</sub>におきかえられる。この場合コンテキスト情報がコードの暗号鍵と一致するかどうかのチェックを行うことはできないが、デバッグの場合にはこのような攻撃の危険を考慮する必要がない。

## 【 0 2 0 5 】

実行コードの演算処理対象であるデータの暗号化は、第 1 実施形態においては図 7 に示す暗号化制御ビット 7 1 3 - 3 によって、第 2 実施形態においては図 1 5 に示す C Y 3 の暗号化制御ビット 1 5 1 7 - 3、および C T 5 9 の暗号化制御ビット 1 5 2 1 - 3 によって制御される。暗号化制御ビットが 0 の時は、デバッグ状態でもデータは暗号化状態で処理される。暗号化制御ビットが 1 にセットされている時は、デバッグ状態の場合に限りデータは平文のまま処理される。

## 【 0 2 0 6 】

本発明ではデータについて、デバッグ状態で暗号化属性のエントリごとに、暗号化と非暗号化を選択できるようにすることで、別のプログラムやデバイスとの間で行われる暗号化されたデータ送受信のデバッグと、プログラムの内部的なデータの平文状態でのデバッグとを、容易に両立させている。

## 【 0 2 0 7 】

特にコードについてはコードを暗号化せず平文として、ページテーブル上のデバッグフラグをセットするだけで、プログラム自体を再コンパイルすることなくデバッグが行なえる利便性を提供している。

## 【 0 2 0 8 】

処理対象データについては、他のプロセスとの通信に使われる領域は、自プロセスはデバッグモードだったとしても、相手のプロセスがデバッグ状態でない限り平文にしておくことはできない場合がある。そこでコードがデバッグモードで動作している場合のみ有効になる暗号化フラグをデータの暗号化属性に設け、このフラグによって暗号化状態を制御できるようにしている。

## 【 0 2 0 9 】

コードのデバッグには、コード自体の暗号状態と、コードを暗号化する鍵の暗号化状態とで、4 通りの組み合わせが考えられる。上述した例は、コード暗号鍵もコードも平文の場合の組み合わせ例である。

## 【 0 2 1 0 】

他のモード（組み合わせ）も考えられる。第 1 が、コード暗号化鍵エントリもコード自体も暗号化された状態で、デバッグモードでコードを実行できるモード

である。第 2 が、コード暗号化鍵エントリは平文の状態、コード自体は暗号化された状態で、デバッグモードでコードを実行できるモードである。第 3 が、コード暗号化鍵エントリは暗号化状態、コードは平文状態の場合である。

#### 【0 2 1 1】

デバッグの目的からはこれらのモードにはあまり意味を持つものではないが、必要に応じてこれらのモードを備えてもよい。

#### 【0 2 1 2】

この際に注意しなければならないのは、第 1 のケースで、暗号化されたコード暗号化鍵エントリに対応する暗号化された実行コードをデバッグモードで実行可能とする場合は、ユーザが暗号化制御のためのビットを操作するだけで、暗号化鍵を知らない保護されたプログラムを、デバッグモードで実行することができてしまう。

#### 【0 2 1 3】

この場合には、デバッグモードビットを不正に操作されないように、デバッグビットをコード暗号化鍵  $K_{code}$  とともに、プロセッサの公開鍵  $K_p$  で暗号化するなどの対策を施して、真のコード暗号化鍵を知らないユーザにデバッグビットを操作されないように注意しなければならない。

#### 【0 2 1 4】

##### 【発明の効果】

以上説明したように、本発明のマイクロプロセッサによれば、マルチタスク環境下で、実行コードと、そのコードの処理対象であるデータの双方を暗号化して保護することにより、オペレーションシステム、あるいは第三者による不正な解析を防止することができる。

#### 【0 2 1 5】

また、データを暗号化して保存した場合の、暗号化属性の不正な書き換えを防止することが可能になる。

#### 【0 2 1 6】

また、処理対象であるデータの暗号化鍵として、固定鍵ではなく任意の乱数  $K_r$  を使用することができ、暗号化されたデータを不正な攻撃から守ることができ

る。

【0217】

また、平文状態でデバッグを行い、不具合が発見された場合は、実行コードの暗号化鍵を知っているプログラムベンダにエラーをフィードバックさせることができる。

【0218】

さらに、暗号化属性情報など、秘密保護に必要な情報にマイクロプロセッサの署名をつけて外部のメモリ上に保存し、必要な部分だけをプロセッサ内部のレジスタに読み込み、読み込み時に署名の検証を行うことで、マイクロプロセッサのメモリの増大を防止し、コストを押さえることができる。この方式では、読み込み時のすり替えに対する安全性も保障される。

【図面の簡単な説明】

【図1】

本発明の第1実施形態にかかるマイクロプロセッサの基本構成を示す図である。

【図2】

図1のマイクロプロセッサの詳細な構成を示す図である。

【図3】

図1のマイクロプロセッサにおけるページディレクトリと、ページテーブル形式を示す図である。

【図4】

図3に示したページテーブルの詳細と、鍵エントリ形式を示す図である。

【図5】

図1のマイクロプロセッサにおけるデータのインタリーブを示す図である。

【図6】

第1実施形態における実行命令コードのアドレス変換から復号化までの情報の流れを示す図である。

【図7】

第1実施形態にかかるマイクロプロセッサのCPUレジスタの構成を示す図で

ある。

【図 8】

第 1 実施形態におけるコンテキスト保存形式を示す図である。

【図 9】

第 1 実施形態における保護ドメイン切り替えの手順を示すフローチャートである。

【図 1 0】

第 1 実施形態における、処理対象であるデータの暗号・復号化の情報の流れを示す図である。

【図 1 1】

本発明のマイクロプロセッサにおける保護ドメイン内の実行制御を示す図である。

【図 1 2】

本発明のマイクロプロセッサにおける保護ドメインから非保護ドメインへの呼び出し、分岐を示す図である。

【図 1 3】

本発明のマイクロプロセッサにおけるデータ暗号化属性共有の手順を示すフローチャートである。

【図 1 4】

データ保護属性の共有に使用されるスレッドテーブルの図である。

【図 1 5】

本発明の第 2 実施形態にかかるマイクロプロセッサの CPU レジスタ構成を示す図である。

【図 1 6】

第 2 実施形態にかかるマイクロプロセッサにおける鍵格納領域レジスタおよび暗号化属性情報特定レジスタと、それぞれ対応するメモリ上の鍵格納領域の関係を示す図である。

【図 1 7】

第 2 実施形態にかかるマイクロプロセッサにおけるコンテキスト保存形式を示

す図である。

【図 18】

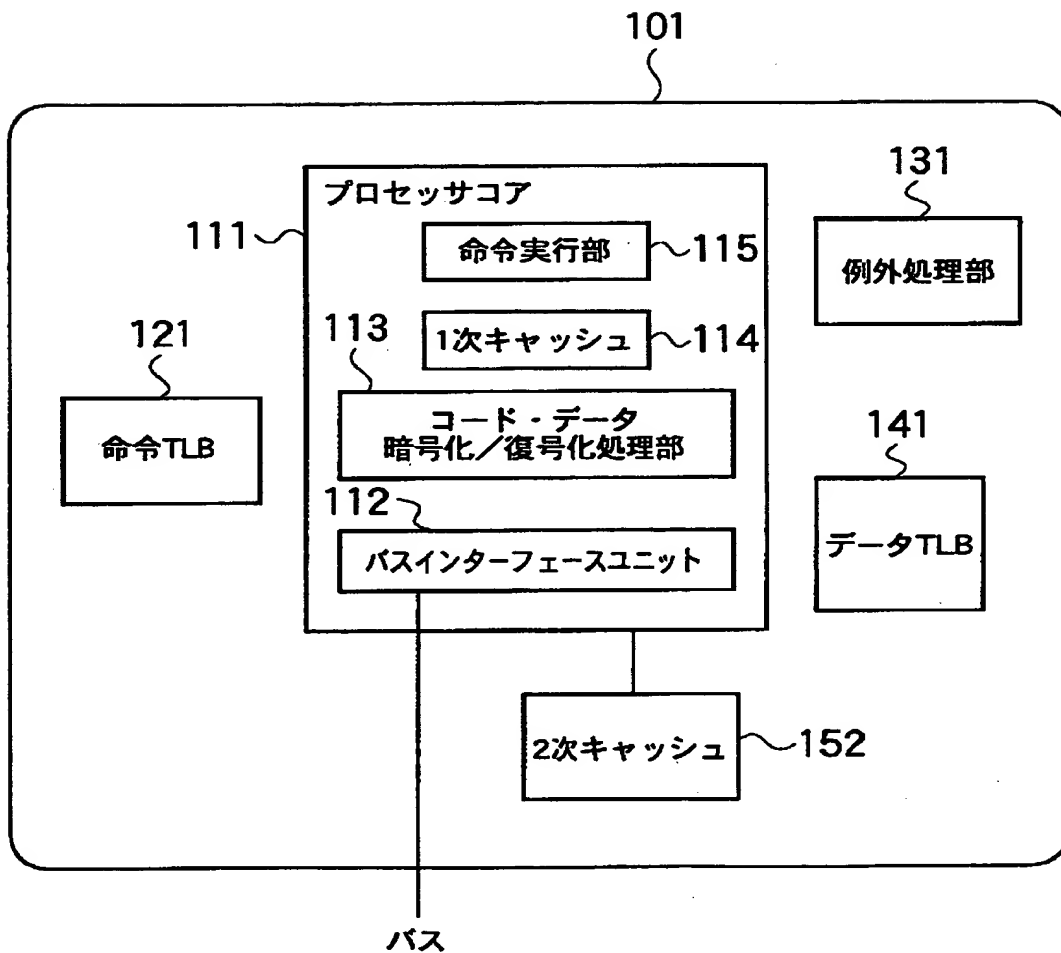
従来技術におけるコンテキスト保存形式を示す図である。

【符号の説明】

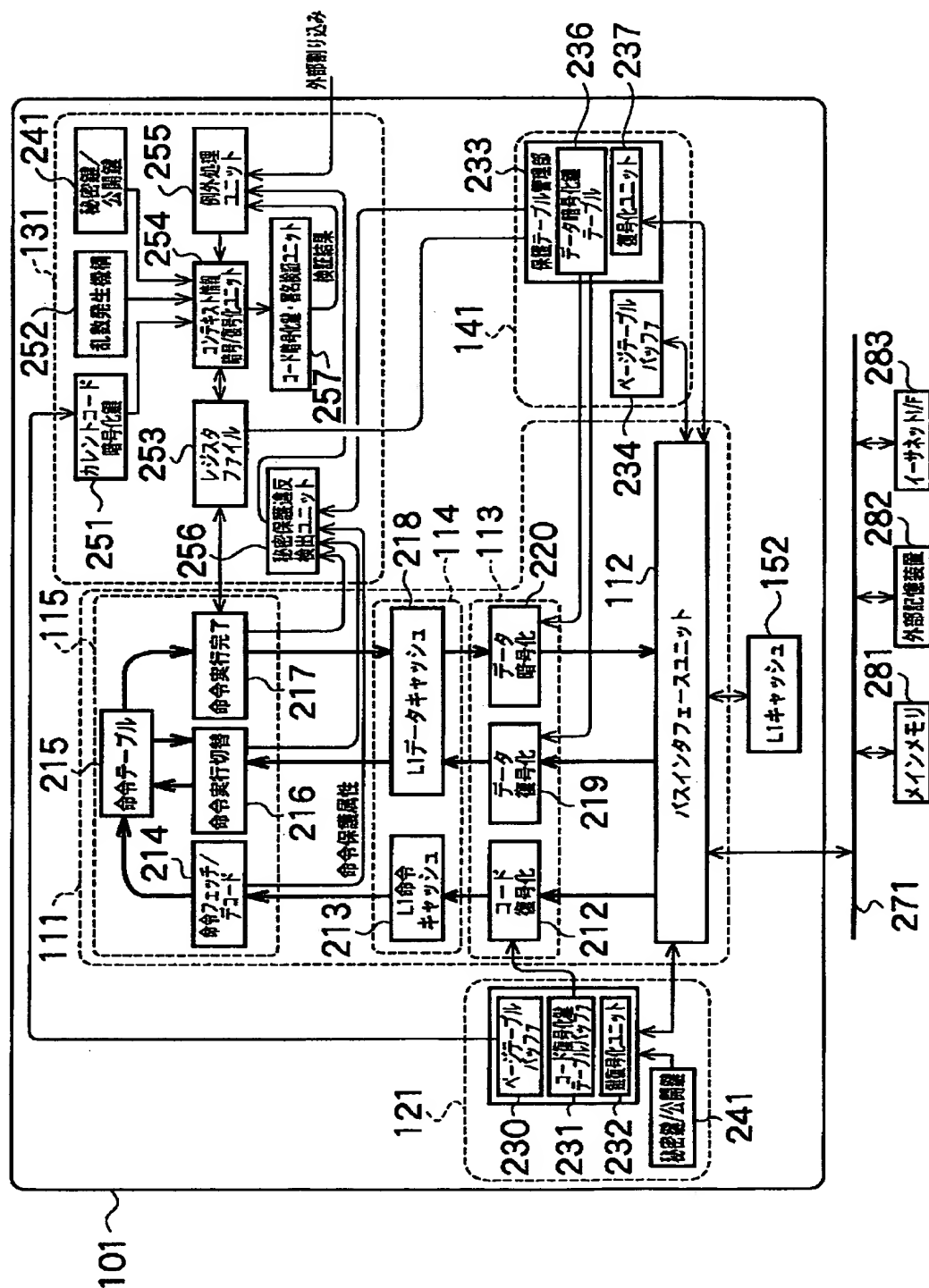
- 101 マイクロプロセッサ
- 111 プロセッサコア
- 112 バスインターフェイスユニット（読み出し手段）
- 113 コード・データ暗号化／復号化処理部
- 114 1次キャッシュ
- 115 命令実行部
- 121 命令TLB
- 131 例外処理部
- 141 データTLB
- 151 2次キャッシュ
- 212 コード復号化ユニット
- 219 データ復号化ユニット
- 220 データ暗号化ユニット
- 233 保護テーブル管理部
- 252 乱数発生機構
- 253 レジスタファイル（プロセッサ外部の記憶手段）
- 254 コンテキスト情報暗号／復号化ユニット
- 257 コード暗号化鍵・署名検証ユニット
- 281 メインメモリ（プロセッサ外部の記憶手段）
- 307、512 ページテーブル
- 309、511 キーテーブル
- CY0～CY3 暗号化属性レジスタ
- CT0～CT59 暗号化属性情報特定レジスタ

【書類名】 図面

【図1】

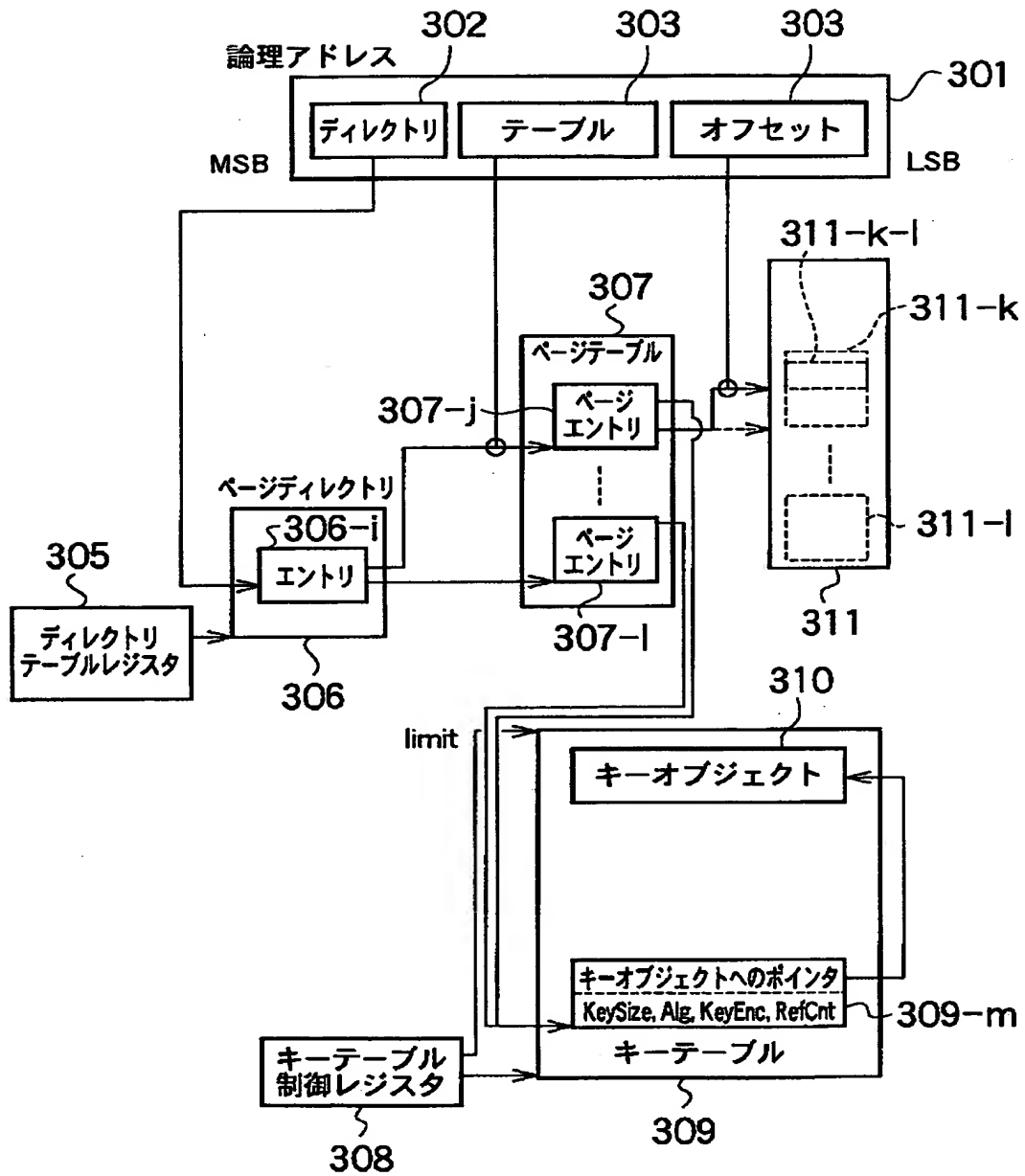


【圖 2】



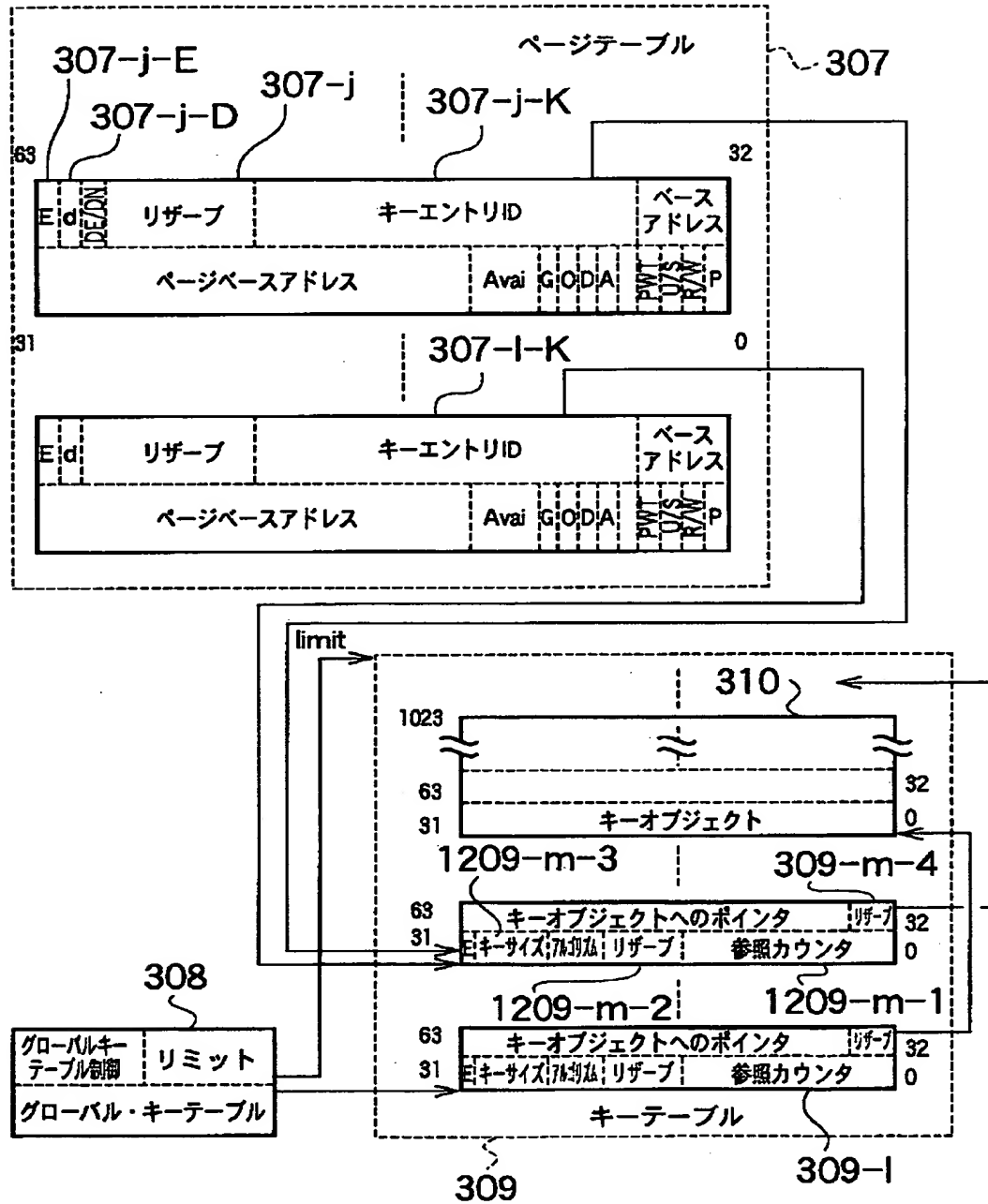


【図 3】



ページディレクトリ/ページテーブル形式

【図 4】



ページテーブル／鍵エントリ形式

【図 5】

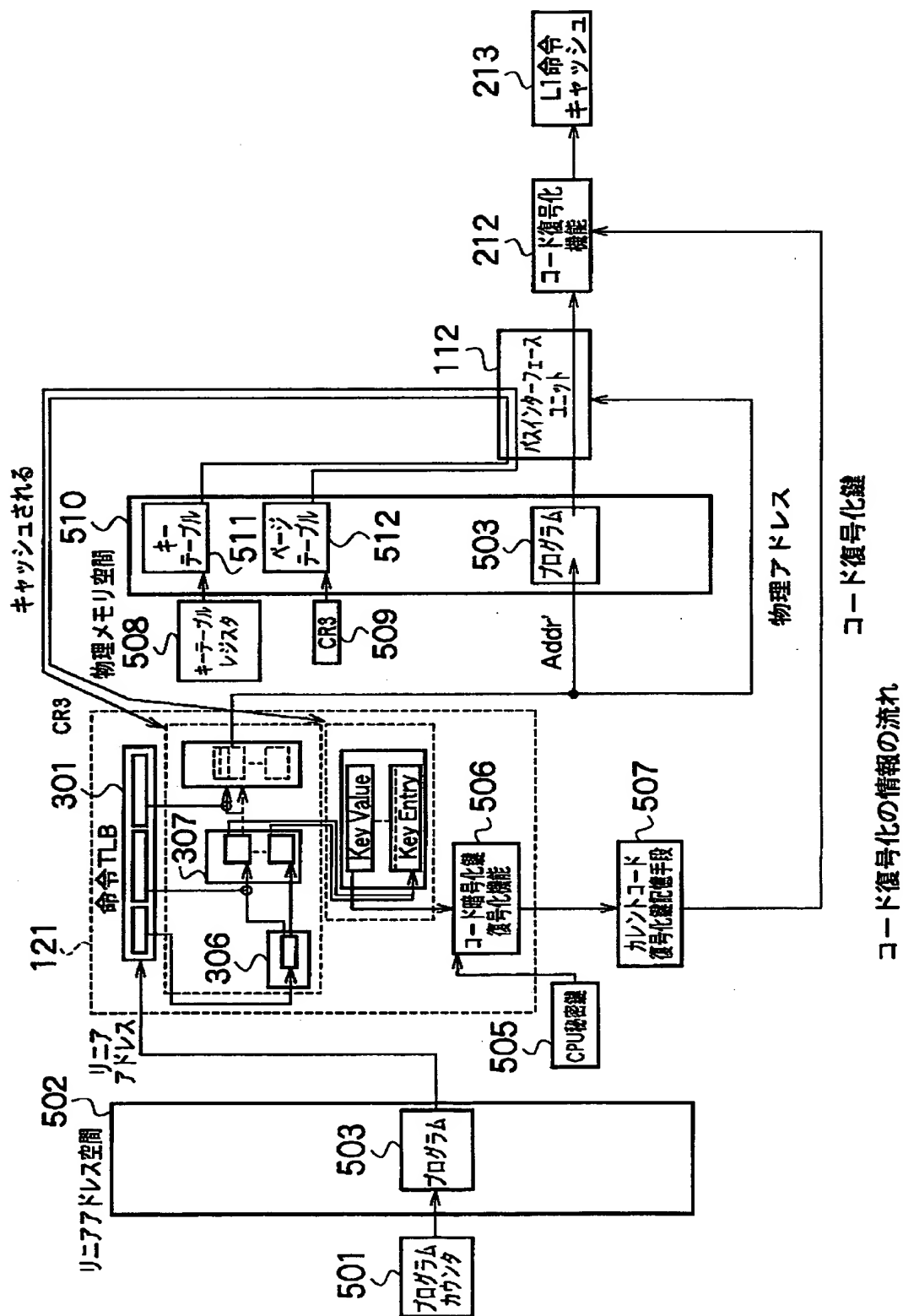
	0	1	2	3
0	A0	A1	A2	A3
1	B0	B1	B2	B3
2	C0	C1	C2	C3
3	D0	D1	D2	D3
4	E0	E1	E2	E3
5	F0	F1	F2	F3
6	G0	G1	G2	G3
7	H0	H1	H2	H3

インタリーブ前

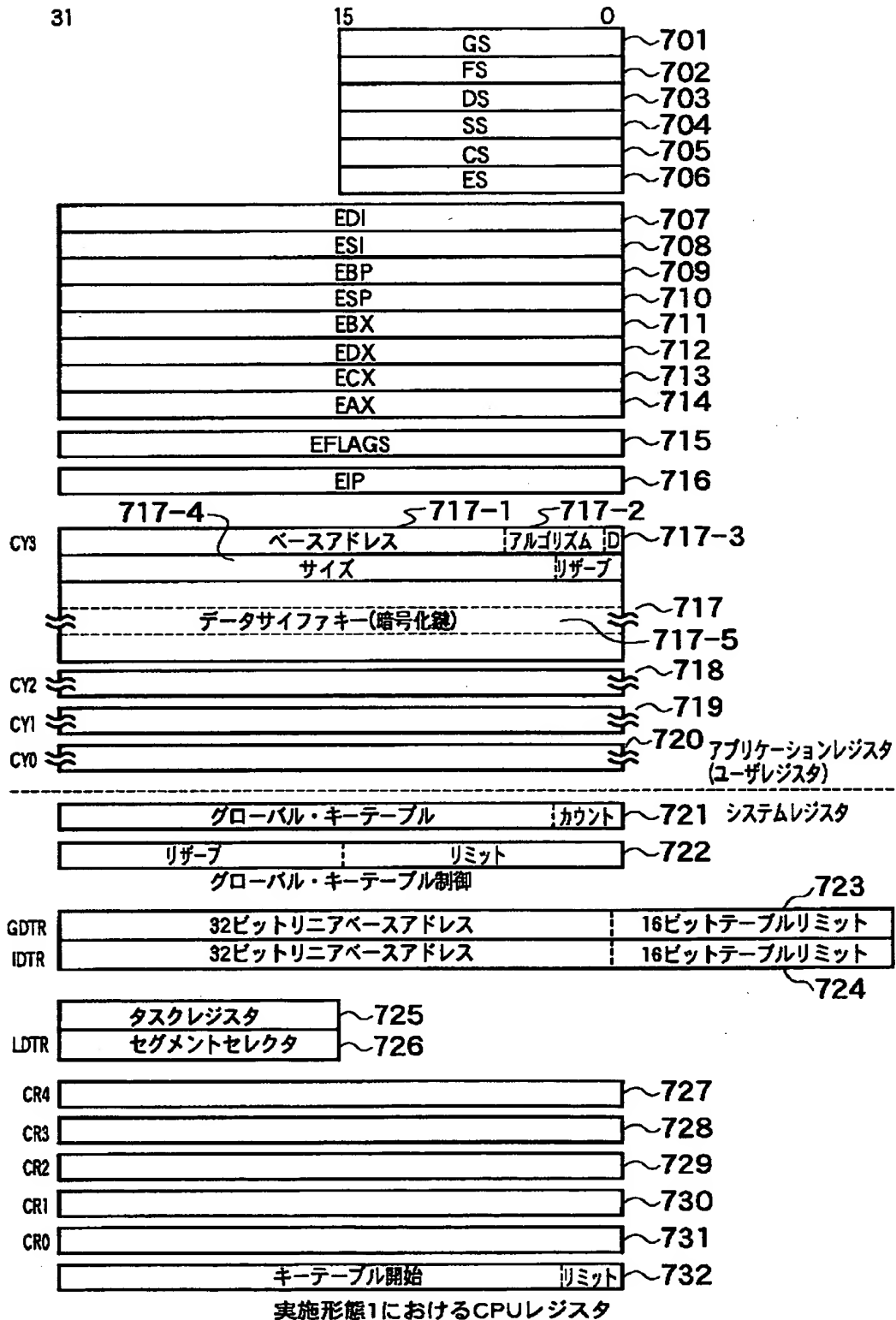
0	A0	B0	C0	D0
1	E0	F0	G0	H0
2	A1	B1	C2	D1
3	E1	F1	G1	H1
4	A2	B2	C2	D2
5	E2	F2	G2	H2
6	A3	B3	C3	D3
7	E3	F3	G3	H3

インタリーブ後

【図 6】



【図 7】

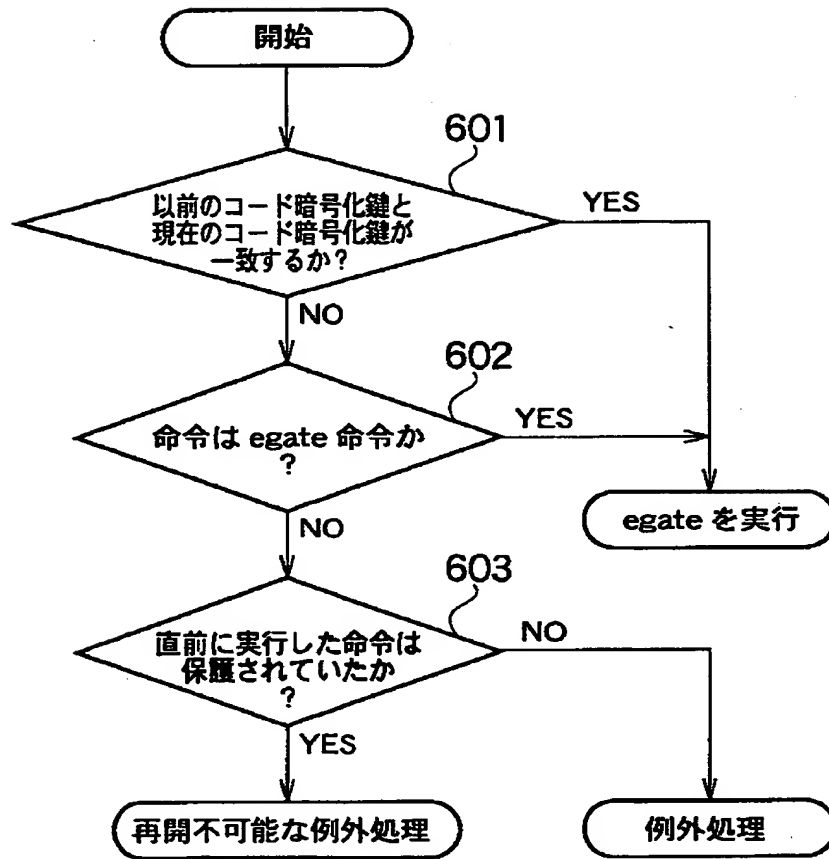


【図 8】

S[messge.] by Ks		834
E[Kr] by Kp		833
E[Kr] by Kcode		832
パディング		831
ベースアドレス	リザーブ	831
サイズ		831
CY3用のデータ暗号化鍵		830
データ暗号化制御レジスタ(CY2)		829
データ暗号化制御レジスタ(CY1)		828
データ暗号化制御レジスタ(CY0)		827
TSSサイズ		826
I/Oマップベースアドレス		825
	LDTセグメントセクタ	825-2
	GS	823
	FS	822
	DS	821
	SS	820
	CS	819
	ES	818
	EDI	817
	ESI	816
	EBP	815
	ESP	814
	EBX	813
	EDX	812
	ECX	811
	EAX	810
	EFLAGS	809
	EIP	808
	CR3(PDBR)	807
	SS2	806
	ESP2	805
	SS1	804
	ESP1	803
	SS0	802
	ESP0	801
	前回のタスクへのリンク	801

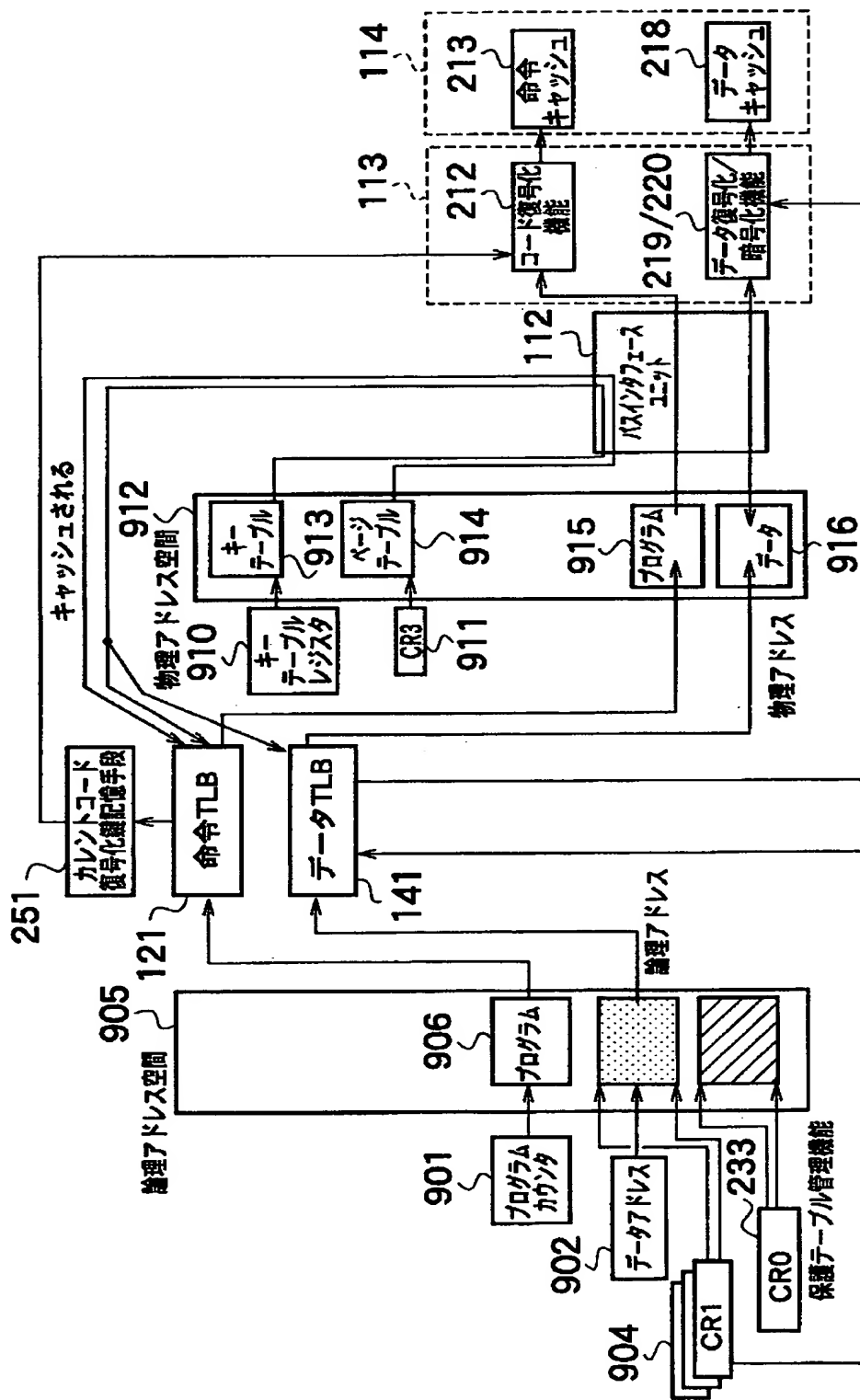
実施形態1におけるコンテキスト保存形式

【図 9】



保護ドメイン切り替えの手順

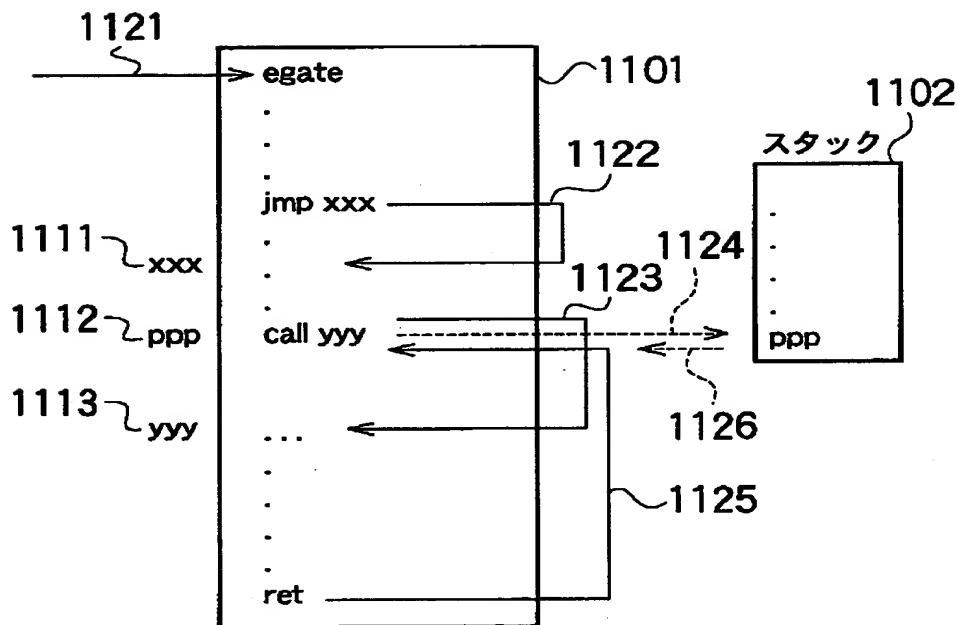
【図 10】



実施例1におけるデータ暗号・復号化の情報の流れ

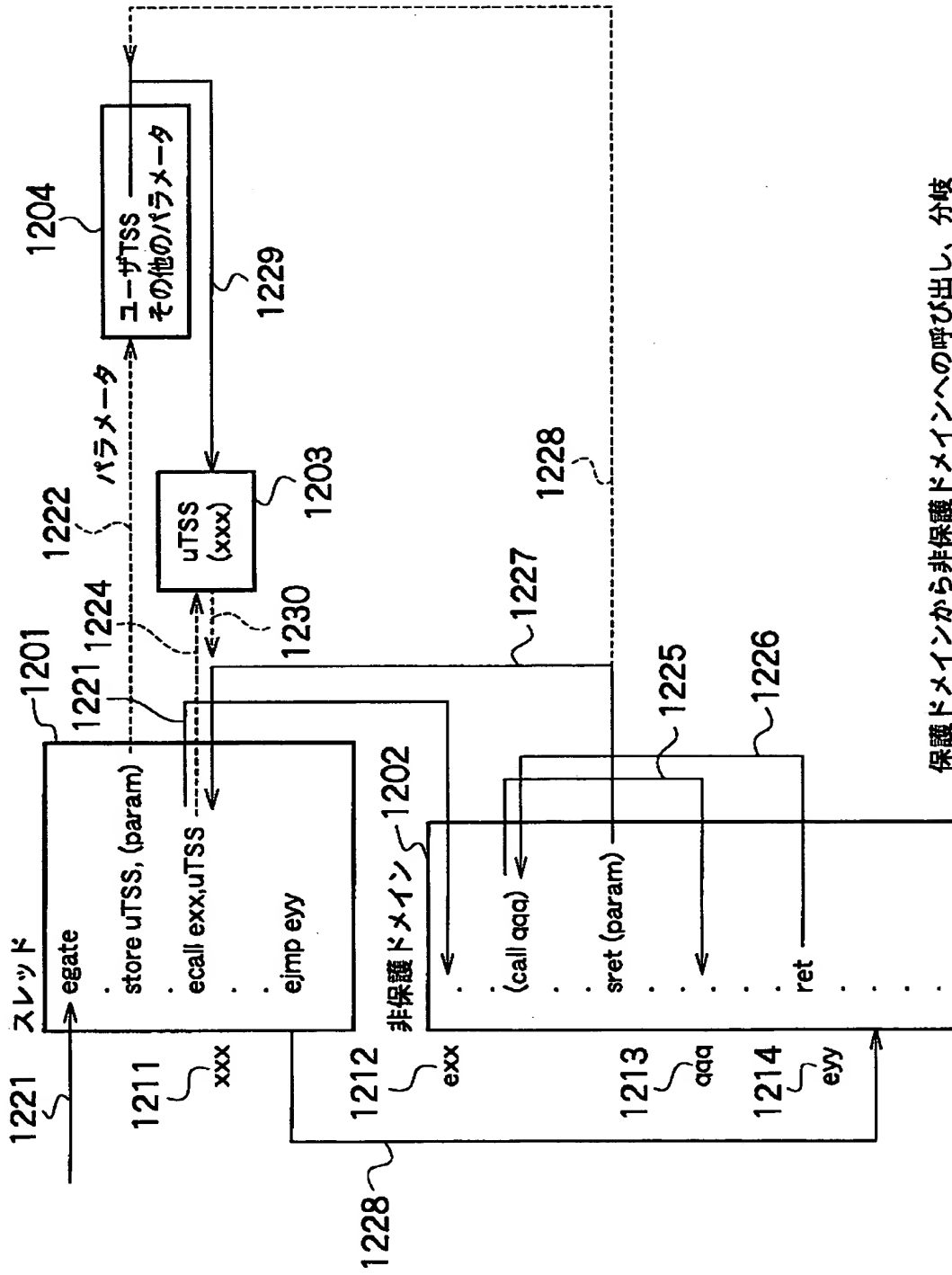


【図 1 1】



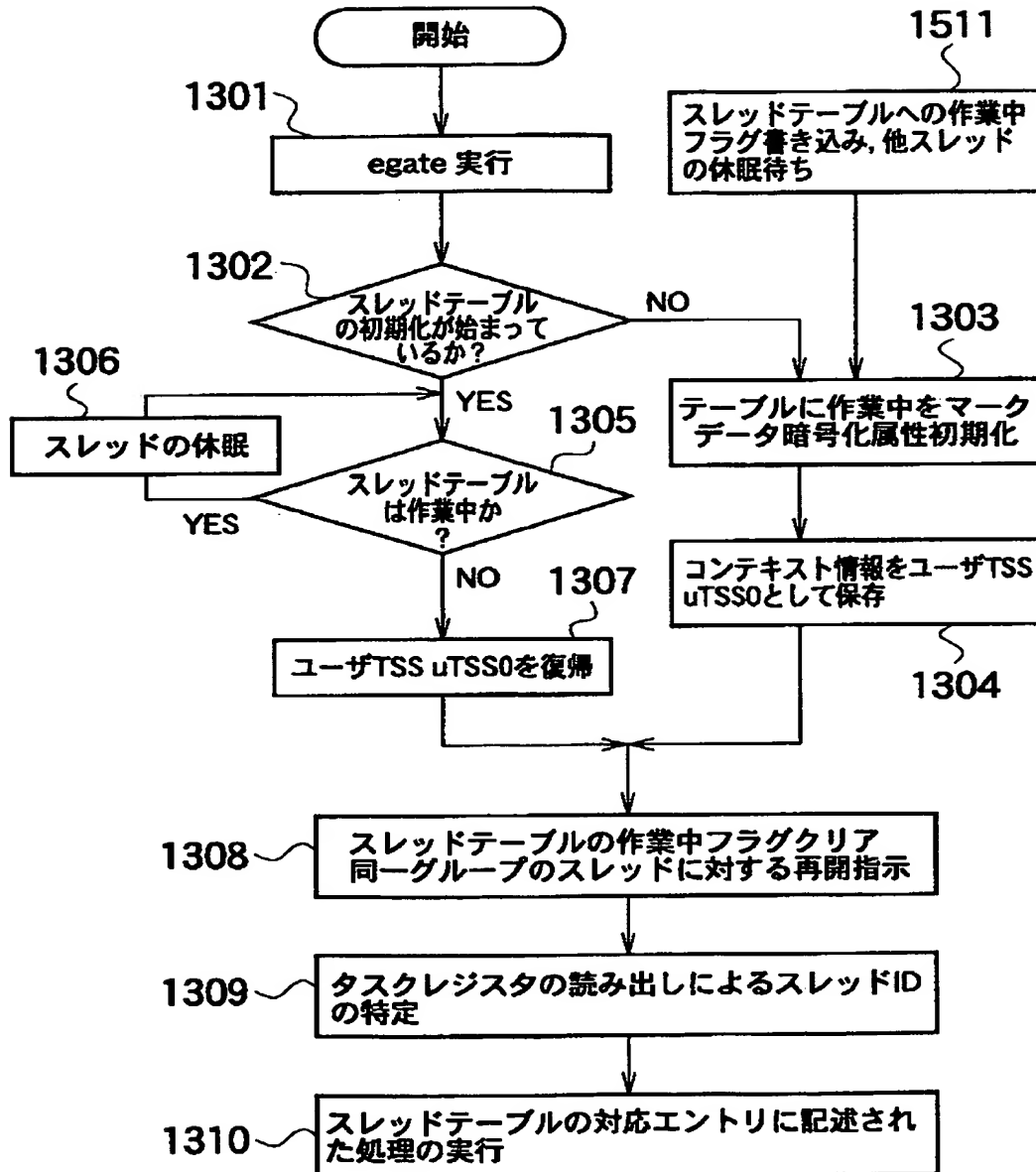
保護ドメイン内の実行制御

【図12】



保護ドメインから非保護ドメインへの呼び出し、分岐

【図 13】



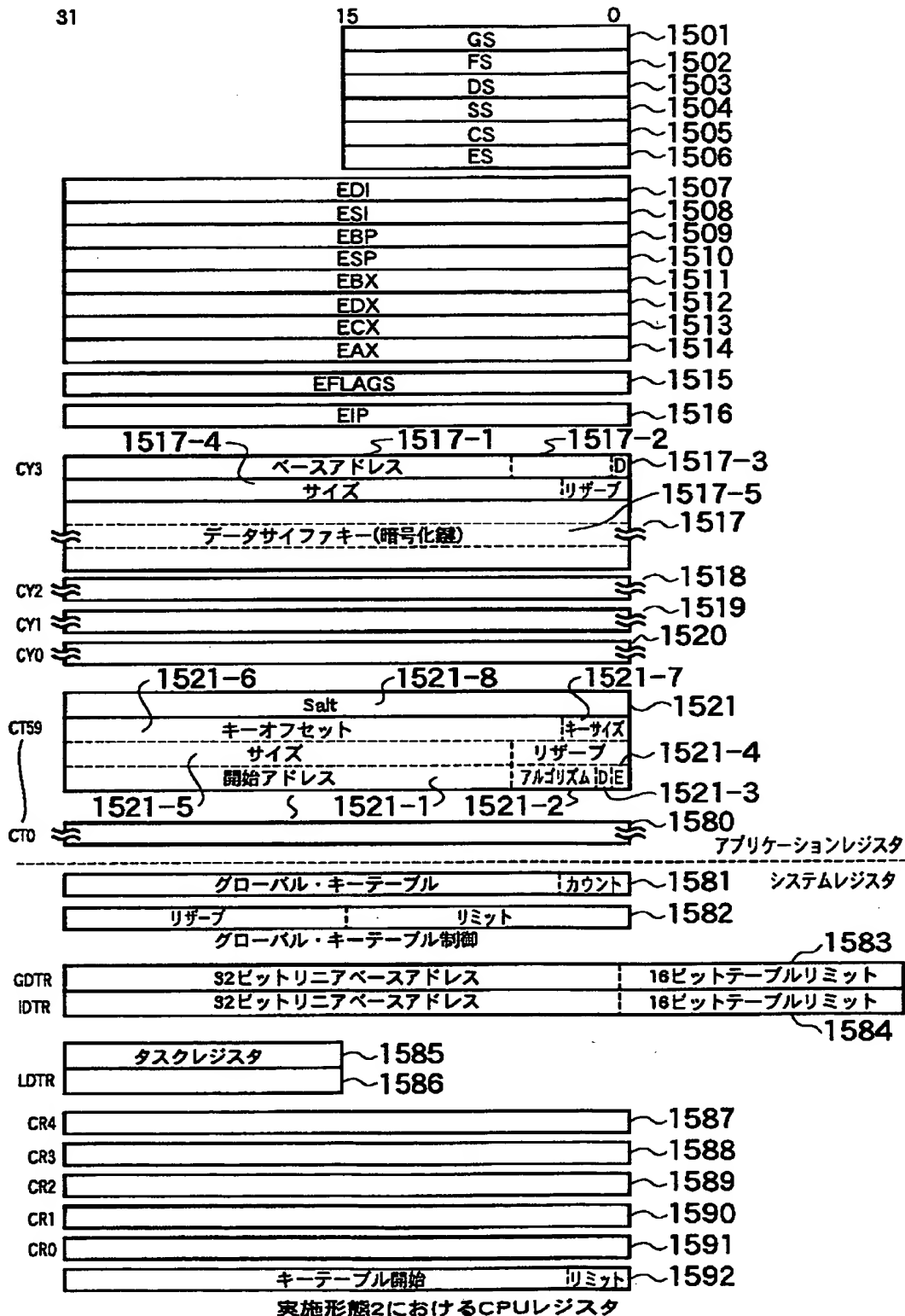
データ保護属性共有手順

【図 1 4】

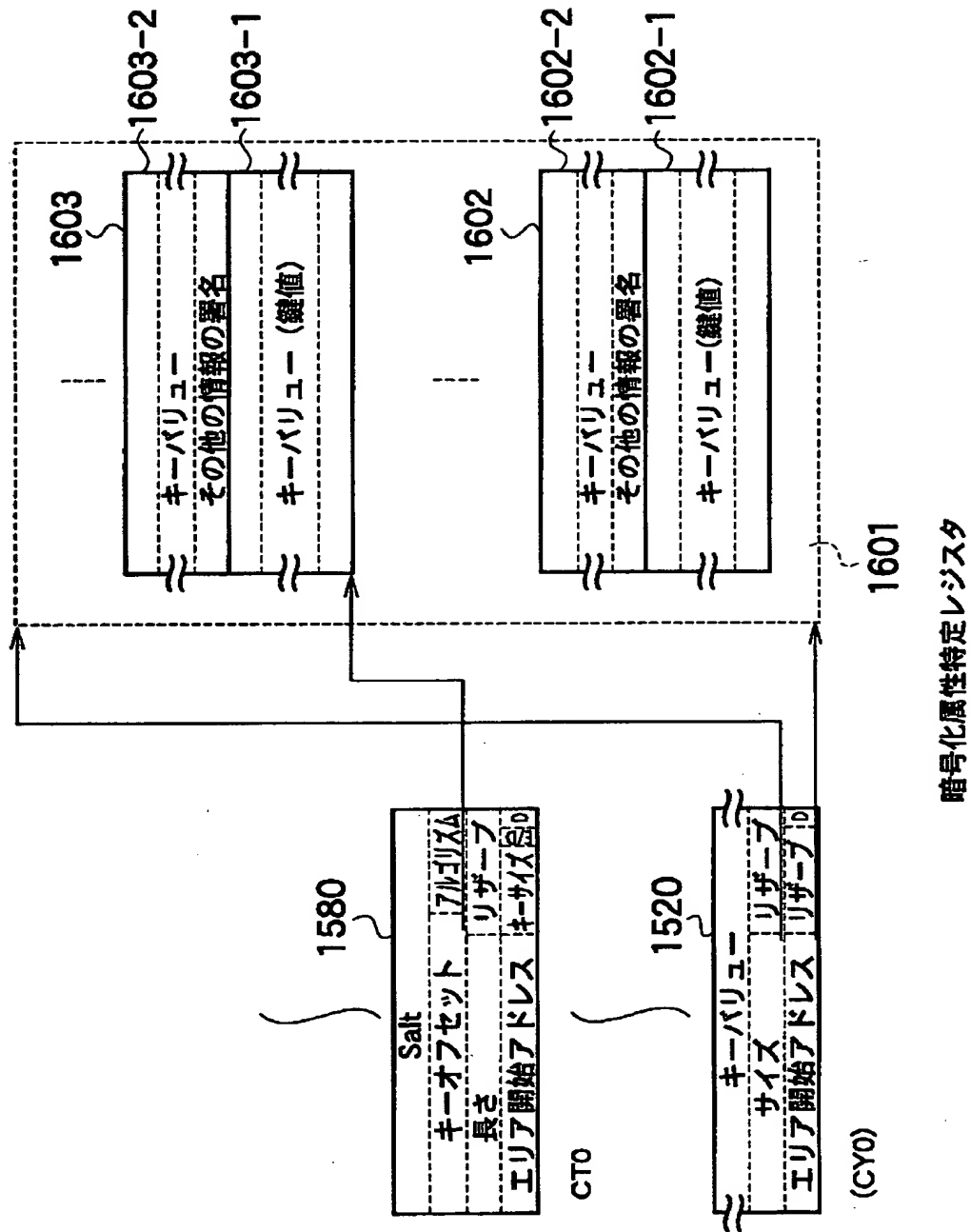
スレッドID	ユーザTSS	パラメータ	作業中フラグ
0	uTSS0	NULL	0
TSS1	uTSS0	pblock1	0
TSS2	uTSS2	pblock2	0
⋮	⋮	⋮	⋮

スレッドテーブル

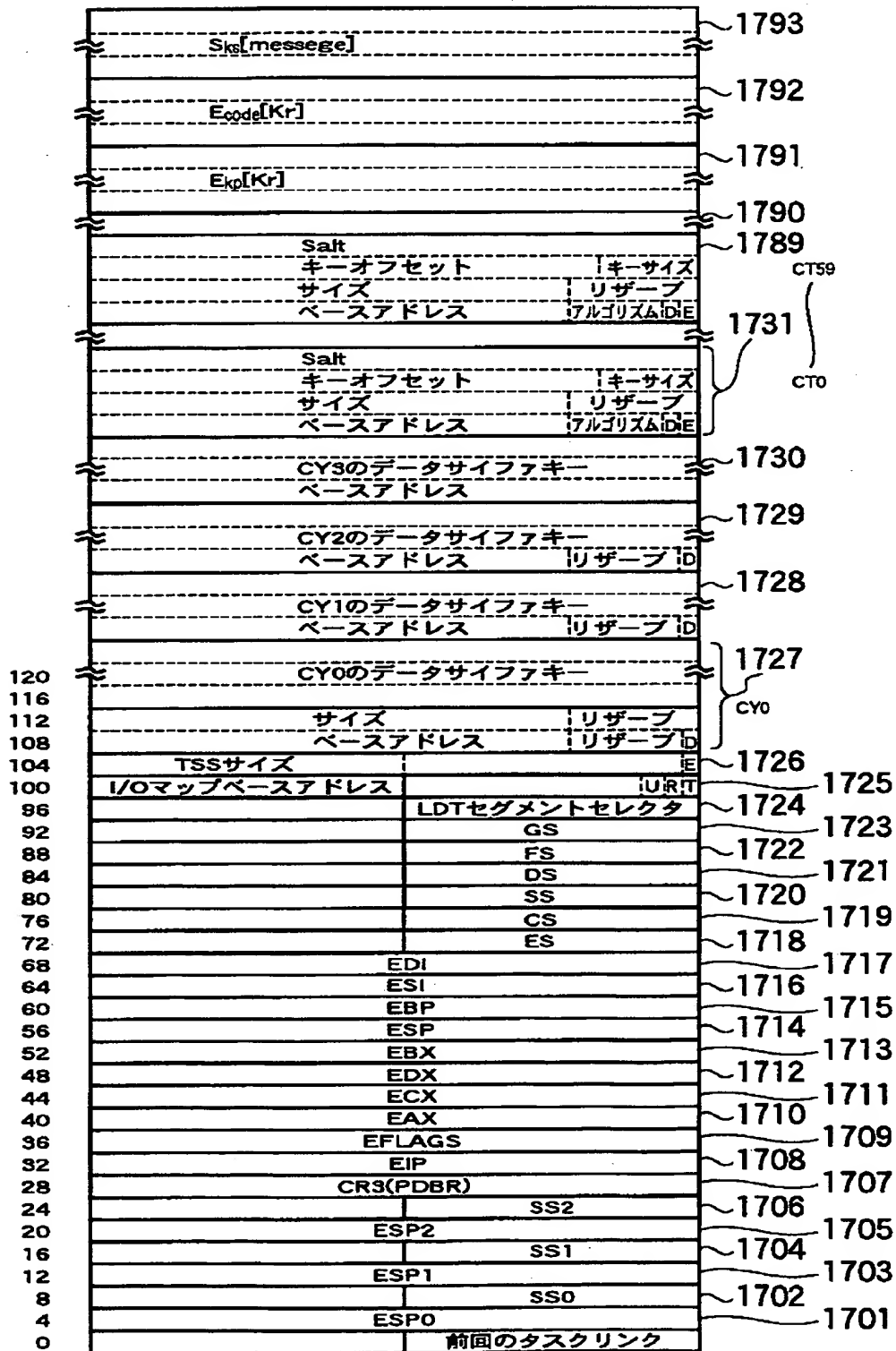
【図15】



【図16】



【図17】



実施形態2におけるコンテキスト保存形式

【図 18】

31	15	0	
I/Oマップベースアドレス		T	100
	LDRセグメントセクタ		96
	GS		92
	FS		88
	DS		84
	SS		80
	CS		76
	ES		72
	EDI		68
	ESI		64
	EBP		60
	ESP		56
	EBX		52
	EDX		48
	ECX		44
	EAX		40
	EFLAGS		36
	EIP		32
	CR3(PDBR)		28
	SS2		24
	ESP2		20
	SS1		16
	ESP1		12
	SS0		8
	ESP0		4
	前回のタスクリンク		0

従来技術におけるコンテキスト保存形式



【書類名】 要約書

【要約】

【課題】 マルチタスク環境下で、暗号化されたプログラムと、そのプログラムが扱うデータの双方を不正な解析から確実に保護することのできるマイクロプロセッサ。

【解決手段】 マイクロプロセッサは、外部のメモリから、それぞれ異なる暗号化鍵により暗号化された複数のプログラムを読み出す。コード復号化ユニットは、読み出したプログラムを対応する復号化鍵で復号化し、命令実行部が復号化されたプログラムを実行する。あるプログラムの実行が中断される場合に、コンテキスト情報暗号／復号化ユニットは、中断されるプログラムのそれまでの実行状態を示す情報と、このプログラムのコード暗号化鍵  $K_{code}$  とを、マイクロプロセッサに固有の公開鍵  $K_p$  で暗号化し、コンテキスト情報としてメインメモリに書き込む。プログラムを再開する場合は、コード暗号化鍵・署名検証ユニットは、マイクロプロセッサの秘密鍵  $K_s$  でコンテキスト情報を復号化し、復号化されたコンテキスト情報に含まれたプログラムの暗号化鍵が、このプログラム本来の暗号化鍵  $K_{code}$  と一致するかどうかを検証する。一致した場合にのみ、プログラムの実行再開を許可する。

【選択図】 図 2

特2000-135010

出 願 人 履 歴 情 報

識別番号 [000003078]

1. 変更年月日	1990年 8月22日
[変更理由]	新規登録
住 所	神奈川県川崎市幸区堀川町72番地
氏 名	株式会社東芝